Effective Stateless Model Checking for C/C++ Concurrency

MICHALIS KOKOLOGIANNAKIS, National Technical University of Athens, Greece
ORI LAHAV, Tel Aviv University, Israel
KONSTANTINOS SAGONAS, Uppsala University, Sweden
VIKTOR VAFEIADIS, Max Planck Institute for Software Systems (MPI-SWS), Germany

We present a stateless model checking algorithm for verifying concurrent programs running under RC11, a repaired version of the C/C++11 memory model without dependency cycles. Unlike most previous approaches, which enumerate thread interleavings up to some partial order reduction improvements, our approach works directly on execution graphs and (in the absence of RMW instructions and SC atomics) avoids redundant exploration by construction. We have implemented a model checker, called RCMC, based on this approach and applied it to a number of challenging concurrent programs. Our experiments confirm that RCMC is significantly faster, scales better than other model checking tools, and is also more resilient to small changes in the benchmarks.

CCS Concepts: • **Theory of computation → Verification by model checking**; Program semantics; • **Software and its engineering → Software testing and debugging**; *Concurrent programming languages*;

Additional Key Words and Phrases: software model checking, weak memory models, C/C++11, RC11

ACM Reference Format:
Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective Stateless Model Checking for C/C++ Concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (January 2018), 32 pages. https://doi.org/10.1145/3158105

1 INTRODUCTION

Suppose we want to verify that a given *multithreaded* program satisfies a given safety specification (e.g., that it never crashes). A standard approach to employ is (bounded) *model checking* [Clarke et al. 1983, 2004; Queille and Sifakis 1982]: explore the set of program states that are reachable from the initial state and determine whether it includes any 'bad' state (i.e., one violating the program's specification). To avoid exploring the same state over and over again, one may straightforwardly just record the set of already visited states. Doing so, however, is often impractical because of the memory required to record this set. This led to *stateless model checking* [Godefroid 1997], which aims to visit all the reachable program states without actually recording them.

Stateless model checking performed naively does not scale because concurrent programs typically have too many interleavings, many of which lead to the same state. This led to techniques, such as *dynamic partial-order reduction* (DPOR) [Flanagan and Godefroid 2005], that cut down some of the redundant explorations.[1] Along this line of work, Abdulla et al. [2014, 2017] have developed

[1] Other competing techniques to DPOR, e.g., MCR [Huang 2015] or techniques based on unfoldings, are discussed in §8. These techniques can in principle explore much fewer states than DPOR, but often yield worse results in practice.

Authors' addresses: Michalis Kokologiannakis, National Technical University of Athens, Greece, michaliskok@softlab.ntua.gr; Ori Lahav, Tel Aviv University, Israel, orilahav@tau.ac.il; Konstantinos Sagonas, Uppsala University, Sweden, kostis@it.uu.se; Viktor Vafeiadis, Max Planck Institute for Software Systems (MPI-SWS), Germany, viktor@mpi-sws.org.

2475-1421/2018/1-ART17
https://doi.org/10.1145/3158105

an "optimal" DPOR technique for interleaving concurrency in the sense that their algorithm never visits two interleavings that are equivalent up to the reordering of independent transitions.

Recently, the DPOR approach was extended to the more realistic settings of *weak memory consistency*, which allow more program behaviors than can be captured by the interleaving semantics with the standard memory representation (as a function from locations to values)—a model commonly called sequential consistency (SC) [Lamport 1979]. In particular, DPOR was extended to the TSO and PSO memory models by Abdulla et al. [2015] and Zhang et al. [2015], but in ways that are still very much based on interleaving semantics. This goes against the definition style of many memory models, such as the C/C++ one [Batty et al. 2011], which is *declarative* (a.k.a. axiomatic). In a declarative semantics, program executions are not represented as traces of interleaved actions but rather as partially ordered graphs, which have to satisfy several consistency constraints. With such a semantics, considering thread interleavings is not only unnecessary, but also harmful for scalability. In fact, as Alglave et al. [2013a] point out, model checking for declarative weak memory models may actually be faster than for SC! Intuitively, this stems from the fact that weak memory declarative semantics tend to be based on local conditions whose violations can be efficiently detected, while SC can only be formulated as a global condition on the whole program execution.

In this paper, we suggest a novel approach for stateless model checking for weak memory concurrency, and apply it to RC11—the repaired version of the C/C++ memory model by Lahav et al. [2017], which corrects a number of problems in the original C/C++ model of Batty et al. [2011]. This allows us to reason about program correctness at the programming language level, and also get correctness at the assembly level via the verified compilation schemes from RC11 to x86, ARM, and POWER. Instead of having the algorithm consider the set of all thread interleavings suitably extended with some reorderings, etc., to account for weak memory behaviors, and then quotient that set by some equivalence relation to avoid redundant explorations, we propose something much simpler: to just *enumerate all consistent execution graphs of a program.*

The main challenge is how to generate all consistent execution graphs of a program without (1) generating any inconsistent graphs, (2) generating the same graph multiple times, and (3) storing the set of graphs already generated. The first two constraints are crucial for performance: in particular, the number of execution graphs of a program is typically much larger than the number of its consistent executions.[2] The third constraint is to avoid excessive memory usage.

The key observation that allows us to only consider consistent graphs is that consistency in RC11—as in most other memory models but not in the Batty et al. [2011] model—is *prefix-closed*. Namely, there exists a partial order R that includes reads-from and (preserved) program order, such that if an execution graph is consistent, then so is every R-prefix of it.[3] Consequently, to generate all consistent graphs, it suffices to start with the initial (empty) graph and extend it gradually by adding one event at a time (in every possible way) and checking for consistency at each step. Since the order in which events are added to a graph is unimportant, to avoid redundant exploration, we fix a certain order (e.g., adding the events of the first thread before those of the second, etc.). This, however, creates another problem, because when a read event is added to the graph, one cannot ensure that all the writes that the read can read from have already been added to the graph.

Accordingly, the second key idea in our algorithm is to extend execution graphs with *revisit sets* capturing the subset of reads whose incoming read-from edges may be changed when a write is added to a graph. When a read is added, the algorithm considers reading from any relevant writes of the existing graph initiating recursive calls to the exploration procedure, and marking the read

[2] This limits the scalability of research tools, such as herd [Alglave et al. 2014], that enumerate all execution graphs of a suitably restricted program and filter out those that are inconsistent.

[3] Batty et al. [2011] allow cycles in program order and reads-from. This results in a number of problems, most importantly the presence of "out-of-thin-air" behaviors [Batty et al. 2015; Boehm and Demsky 2014; Vafeiadis et al. 2015].

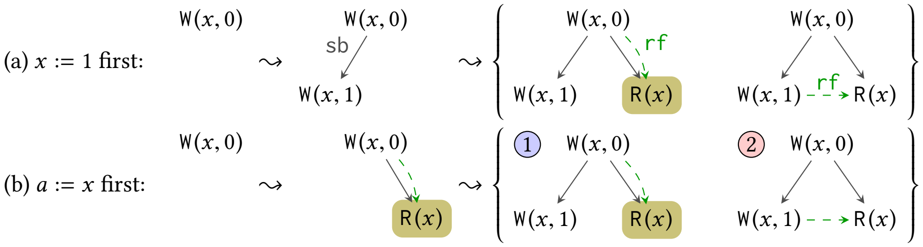

Fig. 1. Two ways of exploring the possible executions of W+R.

as revisitable in exactly one of these calls. Conversely, when a write is added, the algorithm also considers the case where any of the reads in the revisit set could instead read from the new write. A core part of our algorithm concerns the maintenance of the revisit sets in a way that avoids duplication when revisiting existing reads.

Our algorithm handles all the different modes provided by RC11—non-atomics, relaxed, release, acquire, SC—both for accesses and fences. Besides soundness and completeness of our algorithm (i.e., that the algorithm reports a program to be erroneous if and only if the program has some consistent erroneous execution up to some loop unrolling bound), we also prove that our algorithm is *optimal* in the absence of read-modify-write (RMW) accesses and SC atomics. More precisely, we prove that no two sub-explorations of a given RMW-free program could ever visit the same execution; and all explorations of programs without SC atomics result in visiting some RC11-consistent execution of the program.

We have implemented our algorithm in a tool, called RCMC, and applied it to a number of challenging concurrent benchmarks. In §7, we compare our technique with other state-of-the-art stateless model checkers both for SC and weak memory models. The results show that RCMC generally yields lower verification times and scales much better to larger programs. In addition, in comparison to the other tools, RCMC seems to depend much less on the order of threads and on modes of memory accesses. We note that even when RCMC is not optimal (for programs with RMW or SC atomics), our experimental evaluation shows that RCMC runs faster and scales better than other tools.

Outline. The remainder of the paper is structured as follows. In §2, we start with the informal overview of our algorithm with examples. We then review the definition of RC11 from Lahav et al. [2017] (§3), present our model checking algorithm and establish its correctness and optimality properties (§4), and outline a possible simplification of the algorithm for a weakened model (§5). We next briefly present our implementations (§6), evaluate the algorithm's performance (§7), discuss related work (§8), and conclude with some possible future work. The supplementary material for this paper, available at http://plv.mpi-sws.org/rcmc/, contain proofs for all claims in the paper, as well as our implementation.

2 OVERVIEW OF OUR APPROACH

We start by explaining our approach by a sequence of small examples. In all of these examples, x, y, z are global variables, while $a, b, c, \ldots$ are thread-local variables; all global variables are assumed to be zero-initialized by an implicit *main* thread before the other threads are spawned. Unless mentioned otherwise, we assume all of the global variable accesses correspond to RC11 "relaxed" atomic accesses. (Other access modes will be considered in §2.5.)

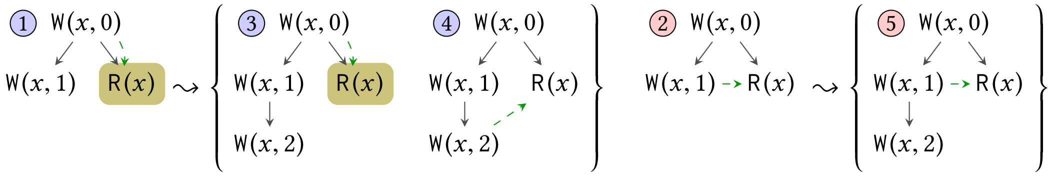

Fig. 2. Additional exploration steps for COWW+R beyond those for W+R.

2.1 Constructing execution graphs and marking nodes revisitable

Our first example is the following tiny program, which has two RC11-consistent executions.

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

The algorithm starts with the initial execution graph containing just the initialization write to x (see Fig. 1(a)). It then adds the $x := 1$ write, which results in an execution graph with two events. The edge between these two events represents their program order: $x := 1$ occurs after the (implicit) initialization write of x. Following RC11, we call such edges sb (sequenced before) edges.

Next, the read of x in the second thread is added. At the moment, there are two possible values it can read (0 and 1) because there are two writes to x in the execution graph. Thus, in the third step, the algorithm generates two graphs to be further explored: one for each case. In the execution graphs, we do not actually record the value returned by the read, because that can be deduced from the write event from whence the read reads, which is displayed as being connected with a dashed “reads-from” edge (rf). In one of the two generated graphs, the newly added read node is highlighted, which means that it may be *revisited* in the future if other writes to x are added, and, as a consequence, its read-from edge may be changed. To avoid redundant revisits, we mark the read as revisitable in *only one* of the generated executions.

To understand the use of the revisitable nodes, consider an alternative run of the algorithm that first adds the read of x, as shown in Fig. 1(b). At this point, the only write that the read could read from is the initialization write. As before, the read is marked as revisitable. When, in the next step, the $x := 1$ write is added, two executions are generated: ① where the read continues to read from the initialization write, and ② where it instead reads from the newly added write. In execution ②, we mark the read as no longer revisitable, again to avoid redundant revisits if another write were to be added to the graph. Note that these two executions are identical to the ones generated by adding the events in the other order. Indeed, crucially, unlike naive operational traces, execution graphs do not expose their generation order.

To understand why the read node is left unmarked in execution ②, consider we extend the program by adding another write at the end of the first thread as follows.

$$\begin{array}{l} x := 1; \\ x := 2 \end{array} \;\left\|\; a := x \right. \qquad \text{(COWW+R)}$$

Further suppose that the $x := 1$ and $a := x$ have been added yielding the two graphs, ① and ②, shown already. Next, the $x := 2$ is to be added. Let’s consider the two executions separately. For execution ①, adding the write node generates two graphs (see Fig. 2): ③ where the read continues to read from the initial write and remains revisitable, and ④ where the read is revisited to read from the W$(x, 2)$ event. For ②, as the read of x is not revisitable, only one execution graph is generated (⑤). If the read of ② were also marked as revisitable, adding the $x := 2$ write to it would also have generated graph ④, leading to redundant exploration as this graph was also generated from ①.

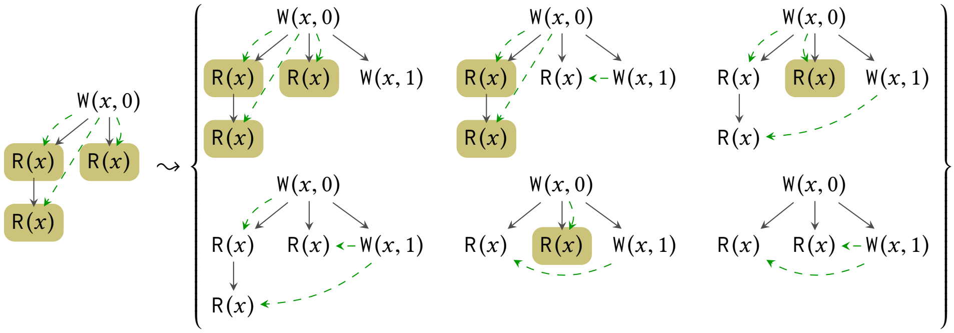

Fig. 3. Key step during the exploration of the possible executions of co2rrw.

2.2 Revisiting multiple reads

We move on to a slightly more advanced example containing multiple reads that need to be revisited when a new write comes along. The following program has six RC11-consistent executions.

$$\begin{array}{l} a := x; \\ b := x \end{array} \;\Bigg\|\; c := x \;\Bigg\|\; x := 1 \qquad \text{(co2rrw)}$$

In the first thread, $a = b = 0$ or $a = 0 \wedge b = 1$ or $a = b = 1$, while in the second thread, $c = 0$ or $c = 1$. (The outcome $a = 1 \wedge b = 0$ is forbidden by the "coherence" property of RC11; intuitively, once the first thread has observed the write of x, it cannot later ignore that write and read from an earlier write of x.) After adding the reads, we reach a configuration where all three reads are revisitable and read from the initialization write (see Fig. 3).

Adding the $x := 1$ write generates six graphs to further explore, also shown in Fig. 3. For every "independent" set of revisitable reads of x, we consider the case where all the reads in the set read from the new write. By independent set, we mean that the set does not contain any nodes that can reach one another. In this case, there are six independent sets—namely, $\emptyset$, $\{c\}$, $\{b\}$, $\{b, c\}$, $\{a\}$, $\{a, c\}$—which give rise to the six graphs. Generally, whenever an event is revisited, we remove all of its ($\mathsf{sb} \cup \mathsf{rf}$)-successors from the graph, because reading from a different write may change the value returned by the read and/or the writes the thread has seen, thereby possibly altering the thread-wise execution of the program after that read. Moreover, to avoid redundant explorations, in each execution we mark the revisited reads and all their predecessors as no longer revisitable.

Finally, the two executions whose $a := x$ read was revisited are extended with a read event corresponding to the $b := x$ read. These new reads can actually only read from the $x := 1$ write, because $x := 1$ was already observed by the preceding $a := x$ read and overwrites the initialization write. Formally, if the $b := x$ read were to read from the initialization write, the execution would be "inconsistent" as it would violate the "coherence" condition of RC11 (formally defined in §3). The algorithm detects this inconsistency before actually producing the inconsistent graph, and therefore does not waste time exploring it or any of its descendants.

The attentive reader may wonder why discarding these inconsistent executions is sound. If such executions *were* produced and further explored, could they eventually lead to some consistent execution? In fact, this cannot happen, because (in the absence of SC atomics) RC11-consistency is *prefix-determined* (Lemma 3.9 in §3.2.6). Roughly speaking, this means that if a read may read from some write after removing an ($\mathsf{sb} \cup \mathsf{rf}$)-suffix of the execution, then it may also read from that write when the suffix is included. (This property does not hold for SC, which forces us to treat SC atomics differently—see §2.5.3.)

2.3 Ruling out incoherent executions

In the previous example, we saw a simple case where reading from a certain write was discarded because it would violate *coherence*. Specifically, after having observed a newer write, a thread cannot read from an older write (e.g., get $a = 1 \land b = 0$ in CO2RRW). More generally, coherence is a property provided by almost all memory models. It ensures that same-location writes are observed by all threads as happening in the same order.

In programs with concurrent writes to the same location, checking for coherence requires a bit more work than we have discussed so far. To illustrate this, consider the following program.

$$x := 1 \;\left\|\; x := 2 \;\right\|\; \begin{array}{l} a := x; \\ b := x \end{array} \;\left\|\; \begin{array}{l} c := x; \\ d := x \end{array}\right. \qquad \text{(CORR2)}$$

Let us focus on the behavior where $a = d = 1 \land b = c = 2$, which is disallowed by RC11 (and any other memory model ensuring coherence), because the two threads observe the writes to x as happening in opposite orders.

To enforce coherence, RC11 executions are extended with a so-called "modification order" (mo). This is a union of total orders—each of which orders the writes to a particular memory location—that is used to impose certain consistency constraints requiring, e.g., that if a thread has observed a certain write, then it cannot later read from an mo-previous write. Thus, to generate all consistent executions, when adding a write to a graph, the exploration algorithm has to generate subexecutions for all the different places the write can be placed in mo. To avoid redundant explorations, special care is required while updating the set of revisitable reads in each subexecution. (We postpone these details to §4.3.)

In §5, we also consider a weaker model that does not fully ensure coherence for non-racy writes. This model, which we call WRC11 (for Weak RC11), does not record the modification order, which simplifies our exploration algorithm and lead to much fewer explored executions in various test cases. For example, in the CORR2 program, recording mo yields 72 executions as opposed to 49 executions under WRC11 (of which 47 are coherent). We refer the reader to §5 for further details about WRC11.

2.4 Handling read-modify-write instructions

We move on to read-modify-write (RMW) instructions, such as compare&exchange (a.k.a. compare&swap or CAS) and fetch&inc (FAI). These instructions combine a read and a write of the same location that are supposed to execute in one atomic step. We will refer to such reads and writes as *exclusive*. Some RMW instructions, such as fetch&inc, always produce two events, while others, such as compare&exchange, always generate the read event and only generate a write event if the read reads an appropriate value. Specifically, $\mathrm{CAS}(x, v, v')$ atomically reads x and if it has value v, replaces it with v' and returns true indicating that the CAS has succeeded (in updating x); otherwise it returns false indicating that the CAS has failed (to update x). The read of the CAS is considered exclusive only when the CAS succeeds. Atomicity of an RMW consisting of an exclusive read r and a write w means that there should not be any other event executed observably between r and w. In particular, two distinct exclusive reads cannot read from the same write event.

Handling RMWs adds a number of subtleties to the algorithm. To see them, let us start with the program shown in Fig. 4 consisting of two parallel atomic increments. After adding the events of the first thread, we reach the execution ⓪ shown in middle of the figure. We use thick edges to denote "RMW pairs" consisting of an exclusive read followed by an exclusive write to the same location. The goal is to generate all consistent executions of the program (i.e., executions ① and ② of Fig. 4 modulo the revisit sets).

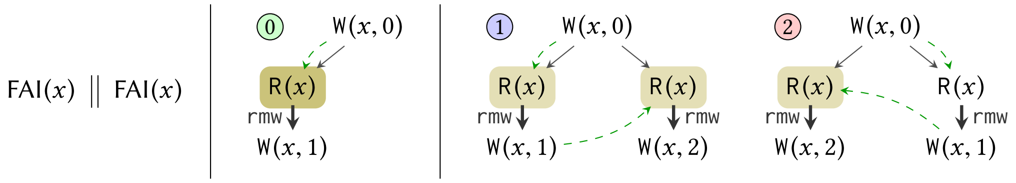

Fig. 4. The FAIS program, an intermediate execution during its exploration, and its final executions.

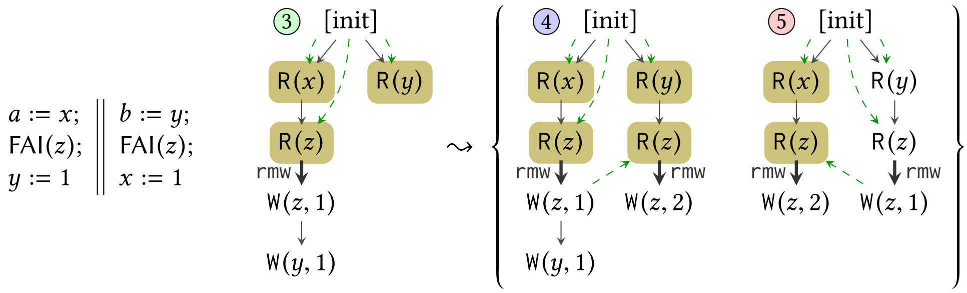

Fig. 5. The LB+FAIS program together with the key step during its exploration.

The next event to add is the read of x from the second thread. This can read from two places—either from the write of the first thread or from the initialization write. In the former case, adding the write of the RMW yields execution ① (ignoring the revisit set for now). When adding this write, the read of the first thread cannot be revisited because it precedes the write in $(\mathsf{sb} \cup \mathsf{rf})^+$.

In the latter case, adding the write event of the RMW leads to an inconsistent execution: both RMWs are reading from the initialization write, thereby violating RMW atomicity. This violation, however, is only temporary: adding the write will lead to another execution, where the read of the first thread is revisited, eventually yielding the execution ② (again, ignoring revisit sets).

The question remaining is what exactly to do with the revisit sets. For soundness, there are two properties that have to be satisfied. First, both executions ① and ② should have the later read (i.e., the one reading from the other RMW) as revisitable. This is needed to cover executions where the second RMW reads from a write after the first RMW (which might appear in some extension of the program with, say, a third thread performing $x := 1$). Second, at least one of the two executions should have the earlier read (i.e., the one reading from the initialization write) as revisitable. This is needed to cover executions where no RMW reads from the initial write, but rather from some write of a third thread. To satisfy these requirements, when adding a read, our algorithm chooses a write that is not read by an RMW as the one where the read is revisitable, and when revisiting a read, if this read is part of an RMW, it keeps it as revisitable. Returning to our example, the algorithm marks both reads of execution ① and the read of the first thread of execution ② as revisitable.

We note that the handling of RMWs, while sound, is not optimal. Consider, for example, an extension of FAIS with a third thread writing $x := 42$. Both executions ① and ② can be revisited. Moreover, in ①, revisiting the read of the first thread will cause the read of the second thread to be removed, and so, among other executions, the algorithm will visit the one where the FAI(x) of the first thread reads 42 and the FAI(x) of the second thread reads 0. This execution, however, is also generated by revisiting ②.

To illustrate in more detail how the revisit sets are calculated, we move on to the LB+FAIS example shown in Fig. 5, which is a variant of the "load buffering" litmus test with increments in the middle of the two threads. Consider execution ③, which is generated by first adding all the events of the first thread and then the read of y from the initial write.

Adding the second $\mathsf{FAI}(z)$ generates executions ④ and ⑤. In ④, the second $\mathsf{FAI}(z)$ reads from the first one; its read is marked as revisitable and all other revisitable reads retain their status. In ⑤, the second $\mathsf{FAI}(z)$ reads from the initialization write. Its read becomes non-revisitable and so do all reads in its ($\mathtt{sb} \cup \mathtt{rf}$)-prefix (i.e., the read of y). Adding the write event of the increment causes the first increment to be revisited and read from the newly added write event. As discussed already, the read of the first thread's increment remains revisitable. Likewise, all reads in its ($\mathtt{sb} \cup \mathtt{rf}$)-prefix (i.e., the read of x) maintain their status. Retaining the revisitability of x is needed to cover the case where the read of x reads from the write of the second thread (to be added subsequently).

A more subtle difficulty arises in programs accessing the same location both using a CAS and using an always succeeding RMW such as FAI. Revisiting a read due to a failed CAS may change its status to that of a successful CAS; that is, the revisited read may turn into an exclusive read. This can conflict with another revisited exclusive read, leading to a temporary state, immediately after revisiting the two reads, where two pending exclusive reads both read from the same write. The conflict is resolved in the next step, when the write of the one exclusive read is added, which will force the other exclusive read to be revisited. An example demonstrating this subtlety is provided in §4.4.

2.5 Handling different access modes and fences

In the examples so far, all memory accesses were treated following the semantics of RC11's *relaxed* atomic accesses. We will now show how the other access modes provided by RC11 are supported.

2.5.1 Release/acquire atomics. We start with release and acquire accesses, which are the simplest kind of accesses to handle. To handle those accesses, RC11 introduces the "*happens before*" order, $\mathtt{hb}$, which is a generalization of the program order. It includes the program order, $\mathtt{sb}$, a number of synchronization patterns, and any transitive combination thereof. The simplest synchronization pattern that induces $\mathtt{hb}$ ordering is when an acquire read operation reads from a release write as in the following "message passing" litmus test.

$$\begin{array}{r||l} x_{\mathtt{rlx}} := 1; & a := y_{\mathtt{acq}}; \\ y_{\mathtt{rel}} := 1 & b := x_{\mathtt{rlx}} \end{array} \tag{MP}$$

The outcome $a = 1 \land b = 0$ is forbidden by RC11 because if $a = 1$, then the read of y *synchronizes with* the write of y, and thus the read of x is aware of the write of x (formally, the write of x happens before the read), which in turn means that the read of x cannot read from the overwritten initialization write. If we were to replace any of the two release/acquire accesses by relaxed accesses, the outcome $a = 1 \land b = 0$ would become allowed, as relaxed accesses on their own do not induce any synchronization. Besides release and acquire accesses, RC11 also has release and acquire fences, whose use in combination with relaxed accesses results in similar synchronization patterns.

To handle release/acquire atomics, our algorithm maintains the happens before order. Whenever a read is added, as before, we consider all the writes that the read could read from without violating RC11's consistency constraints. Similarly, when a write is placed in $\mathtt{mo}$, and when choosing possible reads to revisit after adding a write, we use the calculated $\mathtt{hb}$ relation to avoid violation of RC11's consistency constraints.

2.5.2 Non-atomic accesses. Next, we consider *non-atomic* accesses, which is the default accessing mode in C/C++ programs for variables that have not been declared as atomic. For such accesses, RC11 provides extremely weak semantics. Whenever a program contains a consistent execution with a data race involving non-atomic accesses, the program is deemed to have "undefined behavior"

$$\begin{array}{l||l} x_{\mathtt{sc}} := 1; & y_{\mathtt{sc}} := 1; \\ a := y_{\mathtt{sc}} & b := x_{\mathtt{sc}} \end{array} \qquad (\textsc{sb})$$

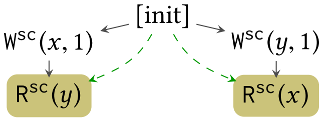

Fig. 6. The “store buffering” program with SC accesses, and its inconsistent execution.

(i.e., be erroneous). Therefore, to handle non-atomic accesses properly, we have to detect data races, i.e., concurrent accesses to the same location, at least one of which being a write, and at least one of which being non-atomic. To avoid redundant checks, it suffices to perform such a check whenever an access is added to the execution graph with respect to accesses already present. Thus, for every full execution of a program, each pair of accesses is checked only once, and moreover these checks are consolidated for all executions having that pair of accesses.

2.5.3 SC atomics. Finally, we consider SC accesses and fences, which can be used to impose strong global ordering between memory accesses, possibly leading to sequential consistency. The standard example demonstrating their use is the “store buffering” litmus test in Fig. 6. RC11 forbids the displayed execution with outcome $a = b = 0$, but allows it for the modified programs where one or more of the accesses are changed to have any non-SC mode. RC11 provides this semantics by defining a relation called psc (“partial SC order”), which it requires to be acyclic in every consistent execution. As psc is not necessarily included in $(\mathtt{sb} \cup \mathtt{rf})^+$, the SC constraint of RC11 makes the model non-prefix-determined (see §2.2). Consequently, one *cannot* simply discard any executions with psc cycles during exploration. For example, suppose we extend the SB program with a third thread performing $y_{\mathtt{sc}} := 2$ (the access mode and value are immaterial) and we add the events from left to right. At some point during the exploration, we will reach the following execution shown to the right above, which is inconsistent according to RC11. If, however, we immediately discard it, we will fail to explore the execution resulting in $a = 2 \wedge b = 0$.

Thus, our algorithm does not check for psc-cycles during exploration. Instead, it checks for the absence of such cycles only before reporting an assertion violation or a forbidden data race. Naturally, this approach induces some redundant explorations, as certain psc-cycles may never be revisited and removed. As an optimization, psc-cycles due to non-revisitable events can be detected eagerly and discarded early.

2.6 Handling spin loops and assume statements

Consider we want to verify a simple test-and-set lock implementation, and we construct a test case comprising of N parallel threads trying to acquire and immediately release a single lock as follows:

$$\begin{array}{l||c||l} \textbf{while } \neg\mathrm{CAS}_{\mathtt{acq}}(x, 0, 1) \textbf{ do skip}; & \cdots & \textbf{while } \neg\mathrm{CAS}_{\mathtt{acq}}(x, 0, 1) \textbf{ do skip}; \\ x_{\mathtt{rel}} := 1 & & x_{\mathtt{rel}} := 1 \end{array}$$

For $N > 1$, the CAS in the acquire loop can fail for an unbounded number of times as the lock may be held by another thread. Even with a moderate loop unrolling bound, explicitly recording these failed loop iterations in the executions can quickly lead to a huge number of executions to explore. Nevertheless, this exploration is completely redundant. Each time a CAS fails in the acquire loop, the thread reaches the exact same local state as if the failed CAS were never performed. Therefore, to get all the possible behaviors of this program, it suffices to consider the case where the CAS succeeds the first time it is performed. A standard way to achieve this is to perform a preprocessing step and convert the program to the following one:

$$\left.\begin{array}{l} a := \mathrm{CAS}_{\mathtt{acq}}(x, 0, 1);\ \textbf{assume}\,(a); \\ x_{\mathtt{rel}} := 1 \end{array}\right\| \cdots \left\| \begin{array}{l} a := \mathrm{CAS}_{\mathtt{acq}}(x, 0, 1);\ \textbf{assume}\,(a); \\ x_{\mathtt{rel}} := 1 \end{array}\right.$$

replacing the CAS loop with its last iteration; namely, with the CAS followed by an **assume** statement reflecting the assumption that CAS succeeds. The question remaining is how to handle such **assume** statements. Clearly, their purpose is to rule out executions where the assumed condition is false. The subtlety is that one cannot always immediately rule out such executions when a false assumed condition is encountered. First, there may be an assertion violation (or a data race) in other (unexplored) threads. Additionally, there may be some revisitable nodes before the assume statement, whose revisiting may, for example, satisfy the assumed condition. For example, consider the exploration of the following program:

$$a := x_{\mathtt{rlx}};\textbf{assume}\ (a \neq 0)\ \|\ x_{\mathtt{rlx}} := 1\ \|\ x_{\mathtt{rlx}} := 2$$

Here, adding the events of the first thread first would yield a revisitable read of x reading the value 0 from the initialization write, and a blocking **assume** statement. In such cases, we cannot stop the exploration, but rather have to continue adding events from other threads to see whether any of the events before the blocking **assume** will be revisited. Here, it would add the $\mathtt{W}^{\mathtt{rlx}}(x, 1)$ event, which would generate a subexploration where the read of x is revisited. The exploration where the read is not revisited is then further explored so as to consider the case where it will be revisited by the write of the third thread. When, finally, all threads have either finished or are blocked (due to a failed assumption), we can discard the execution.

3 THE RC11 MEMORY MODEL

In this section, we present our formal programming language semantics, following the RC11 memory model [Lahav et al. 2017]. This is done in three stages. First, in §3.1, we define the representation of programs. Then, in §3.2, we define execution graphs and RC11-consistency. Finally, in §3.3, we connect these two notions together and define when a program is erroneous.

3.1 Sequential programs

For simplicity, we assume that, after preprocessing, programs are of the form $\|_{i \in \mathsf{Tid}}\ P_i$, where $\mathsf{Tid} = \{1, \ldots, N\}$ is a set of thread identifiers, and each P_i is a sequential loop-free deterministic program. We also assume finite sets Loc and Val of locations and values. The set of accesses and fences modes is given by $\mathsf{Mod} \triangleq \{\mathtt{na}, \mathtt{rlx}, \mathtt{acq}, \mathtt{rel}, \mathtt{acqrel}, \mathtt{sc}\}$, and is partially ordered as follows:

$$\mathtt{na} \sqsubset \mathtt{rlx} \sqsubset \mathtt{acq} \sqsubset \mathtt{acqrel} \sqsubset \mathtt{sc} \qquad \text{and} \qquad \mathtt{rlx} \sqsubset \mathtt{rel} \sqsubset \mathtt{acqrel}.$$

To refrain from setting a concrete syntax, we represent the sequential programs P_i as functions that return the *label* of the next action to execute given a (partial) program trace.

Definition 3.1. A *label* takes one of the following forms:

- read label: $\mathtt{R}^o(x, V)$ where $x \in \mathsf{Loc}$, $V \subseteq \mathsf{Val}$, and $o \in \mathsf{Mod}_{\mathtt{R}} \triangleq \{\mathtt{na}, \mathtt{rlx}, \mathtt{acq}, \mathtt{sc}\}$. The set V stores all *exclusive values*: reading of which will make the read exclusive.
- write label: $\mathtt{W}^o(x, v)$ where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, and $o \in \mathsf{Mod}_{\mathtt{W}} \triangleq \{\mathtt{na}, \mathtt{rlx}, \mathtt{rel}, \mathtt{sc}\}$.
- fence label: $\mathtt{F}^o$ where $o \in \mathsf{Mod}_{\mathtt{F}} \triangleq \{\mathtt{acq}, \mathtt{rel}, \mathtt{acqrel}, \mathtt{sc}\}$.
- error label: $\mathtt{error}$.
- blocking label: $\mathtt{block}$.

We denote the set of all labels by Lab. The functions $\mathtt{typ}$, $\mathtt{mod}$, $\mathtt{loc}$, $\mathtt{val}$, and $\mathtt{exvals}$ return (when applicable) a label's type ($\mathtt{R}/\mathtt{W}/\mathtt{F}/\mathtt{error}/\mathtt{block}$), mode, location, written value, and exclusive values.

```
$x_{\mathtt{rlx}}$ := 1;
$a$ := $y_{\mathtt{sc}}$;
assume ($a \in \{1, 2, 3\}$);
if $a = 1$ then
  $b$ := $\mathrm{CAS}_{\mathtt{acq}}(x, 0, 1)$;
else
  assert ($a = 2$);
  $y_{\mathtt{na}}$ := 4;
```

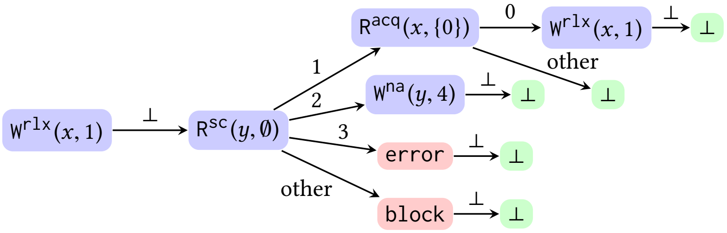
$\mathtt{W}^{\mathtt{rlx}}(x, 1)$
⊥
$\mathtt{R}^{\mathtt{sc}}(y, \emptyset)$
1
2
3
other
$\mathtt{R}^{\mathtt{acq}}(x, \{0\})$
0
$\mathtt{W}^{\mathtt{rlx}}(x, 1)$
⊥
⊥
other
⊥
$\mathtt{W}^{\mathtt{na}}(y, 4)$
⊥
⊥
error
⊥
⊥
block
⊥
⊥

Fig. 7. Representation of a sequential program as a labeled automaton.

Definition 3.2. A *trace* is a finite sequence of Val $\uplus$ $\{\bot\}$. We denote by Trace the set of all traces. A sequential program is represented as a partial function P_i : Trace $\rightharpoonup$ Lab.

To illustrate this representation, consider the program in Fig. 7 and its depiction as a labeled automaton. Each state is labeled with the label that is generated by that state, while the edges are labeled with the possible values that the state may return. In particular, states labeled with $\bot$ are terminal states, states labeled with a read label have multiple next states—one for each value the read could return—and states labeled with other labels have exactly one next state. Reading this automaton as a function from traces to labels is straightforward. For example, $P_i(\langle \bot, 1 \rangle) = \mathtt{R}^{\mathtt{acq}}(x, \{0\})$. Note that:

- The programming language semantics ensures that the next event of an exclusive read is its corresponding exclusive store, that is: whenever $P_i(t) = \mathtt{R}^{o_{\mathtt{R}}}(x, V)$ and $v_{\mathtt{R}} \in V$, we have $P_i(t \cdot \langle v_{\mathtt{R}} \rangle) = \mathtt{W}^{o_{\mathtt{W}}}(x, v_{\mathtt{W}})$ for some $o_{\mathtt{W}}, v_{\mathtt{W}}$ such that $\langle o_{\mathtt{R}}, o_{\mathtt{W}} \rangle \in \mathsf{Mod}_{\mathtt{RMW}}$, where:
$$\mathsf{Mod}_{\mathtt{RMW}} \triangleq \{\langle \mathtt{rlx}, \mathtt{rlx} \rangle, \langle \mathtt{acq}, \mathtt{rlx} \rangle, \langle \mathtt{rlx}, \mathtt{rel} \rangle, \langle \mathtt{acq}, \mathtt{rel} \rangle, \langle \mathtt{sc}, \mathtt{sc} \rangle\}$$
These pairs stand for relaxed, acquire, release, acquire-release, and sequentially consistent RMW operations, respectively.
- Error and blocking labels block the execution: if $P_i(t) \in \{\mathtt{error}, \mathtt{block}\}$, then $P_i(t \cdot \langle \bot \rangle) = \bot$.

3.2 Execution graphs

Next, we formally introduce execution graphs and the RC11-consistency constraints.

Before we start, we introduce some notation. Given a binary relation R, $dom(R)$ and $codom(R)$ denote its domain and codomain. Further, we write $R^?$, R^+, and R^* respectively to denote its reflexive, transitive, and reflexive-transitive closures. The inverse relation is denoted by R^{-1}. We denote by $R_1; R_2$ the left composition of two relations R_1, R_2, and assume that ; binds tighter than $\cup$ and $\setminus$. We denote by $[A]$ the identity relation on a set A. In particular, $[A]; R; [B] = R \cap (A \times B)$. We omit the set parentheses when writing expressions like $[a_1, \dots, a_n]$ (which stands for $\{\langle a_1, a_1 \rangle, \dots, \langle a_n, a_n \rangle\}$). Finally, given an element a, $\mathsf{succ}_R(a)$ denotes the unique element b such that $\langle a, c \rangle \in R \Leftrightarrow \langle b, c \rangle \in R^?$ for every c (undefined if such an element does not exist or is not unique).

Definition 3.3. An *event* is a tuple $\langle i, n, l \rangle \in (\mathsf{Tid} \cup \{0\}) \times \mathbb{N} \times \mathsf{Lab}$, where i is a thread identifier (0 for initialization events), n is a serial number inside each thread, and l is a label. The functions tid and lab return the thread identifier and the label of an event. In addition, the functions typ, mod, loc, val, and exvals are extended to events in the obvious way. We denote the set of all events by E, and use $\mathsf{R}, \mathsf{W}, \mathsf{F}, \mathtt{error}, \mathtt{block}$ to denote the set of events of the respective type. We use subscripts and superscripts to denote the accessed location, the thread identifier, and the mode (e.g., $\mathsf{E}^i = \{a \in \mathsf{E} \mid \mathtt{tid}(a) = i\}$ and $\mathsf{W}_x^{\sqsupseteq \mathtt{rel}} = \{w \in \mathsf{W} \mid \mathtt{loc}(w) = x \ \wedge \ \mathtt{mod}(w) \sqsupseteq \mathtt{rel}\}$).

Definition 3.4. The set E_0 of *initialization events* is given by $\mathsf{E}_0 \triangleq \{\langle 0, 0, \mathtt{W}^{\mathtt{na}}(x, 0) \rangle \mid x \in \mathsf{Loc}\}$.

Definition 3.5. The *sequenced-before* relation, denoted $\mathtt{sb}$, is given by:

$$\mathtt{sb} \triangleq \mathsf{E}_0 \times (\mathsf{E} \setminus \mathsf{E}_0) \cup \{\langle\langle i_1, n_1, l_1\rangle, \langle i_2, n_2, l_2\rangle\rangle \in (\mathsf{E} \setminus \mathsf{E}_0) \times (\mathsf{E} \setminus \mathsf{E}_0) \mid i_1 = i_2 \land n_1 < n_2\}$$

Definition 3.6. An *execution* G is a tuple $\langle E, rf, mo\rangle$ where:

1. E is a finite set of events containing the set E_0 of initialization events.
2. rf, called *reads-from*, is a binary relation on E satisfying:
   - $w \in \mathtt{W}$, $r \in \mathsf{R}$ and $\mathtt{loc}(w) = \mathtt{loc}(r)$ for every $\langle w, r\rangle \in rf$.
   - $w_1 = w_2$ whenever $\langle w_1, r\rangle, \langle w_2, r\rangle \in rf$ (that is rf^{-1} is a partial function).
3. mo, called *modification order*, is a disjoint union of relations $\{mo_x\}_{x \in \mathsf{Loc}}$, such that each mo_x is a strict partial order on $E \cap \mathtt{W}_x$.

We denote the components of an execution $G = \langle E, rf, mo\rangle$ by $G.\mathsf{E}$, $G.\mathtt{rf}$ and $G.\mathtt{mo}$. In addition, G induces the following sets, functions and relations:

- $G.\mathtt{val}_\mathsf{r} : \mathsf{E} \rightharpoonup \mathsf{Val}$ returns the value read by event r, that is: $G.\mathtt{val}_\mathsf{r}(r) = \mathtt{val}(w)$ where w is the (unique) write such that $\langle w, r\rangle \in rf$. If r is not a read event or such a write does not exist, then $G.\mathtt{val}_\mathsf{r}(r) = \bot$.
- $G.\mathtt{sb}$ is given by $G.\mathtt{sb} \triangleq [G.\mathsf{E}]; \mathtt{sb}; [G.\mathsf{E}]$.
- A read event $r \in E$ is called *exclusive* in G if $G.\mathtt{val}_\mathsf{r}(r) \in \mathtt{exvals}(r)$. We write $G.\mathsf{R}^{\mathsf{ex}}$ to denote the set of exclusive reads in G.
- $G.\mathtt{rmw}$, called *read-modify-write pairs*, is the relation given by $G.\mathtt{rmw} = [G.\mathsf{R}^{\mathsf{ex}}]; G.\mathtt{sb}|_{\mathrm{imm}}$, where $\mathtt{sb}|_{\mathrm{imm}} = \mathtt{sb} \setminus (\mathtt{sb}; \mathtt{sb})$. (Note that for executions resulting from programs we have $w \in \mathtt{W}$, $\mathtt{loc}(r) = \mathtt{loc}(w)$ and $\langle \mathtt{mod}(r), \mathtt{mod}(w)\rangle \in \mathsf{Mod}_{\mathtt{RMW}}$ for every $\langle r, w\rangle \in G.\mathtt{rmw}$.)

The main part of the memory model is filtering the consistent executions among all executions of the program by imposing certain constraints.

3.2.1 Completeness. The first constraint is very simple: every read should read a value. Accordingly, we call an execution G is called *complete* if $G.\mathsf{E} \cap \mathsf{R} \subseteq codom(G.\mathtt{rf})$.

3.2.2 Coherence. Coherence (a.k.a. SC-per-location) requires that, for every particular location, all threads agree on the order of accesses to that location. Moreover, this order should be consistent with the *happens-before* order ($\mathtt{hb}$), which intuitively records when an event is globally perceived as occurring before another one. To define $\mathtt{hb}$, several derived relations are needed:

$$G.\mathtt{rseq} \triangleq \textstyle\bigcup_{x \in \mathsf{Loc}} [\mathtt{W}_x]; G.\mathtt{sb}^?; [\mathtt{W}_x^{\sqsupseteq \mathtt{rlx}}]; (G.\mathtt{rf}; G.\mathtt{rmw})^* \qquad (\textit{release sequence})$$

$$G.\mathtt{sw} \triangleq [\mathsf{E}^{\sqsupseteq \mathtt{rel}}]; ([\mathsf{F}]; G.\mathtt{sb})^?; G.\mathtt{rseq}; G.\mathtt{rf}; [\mathsf{R}^{\sqsupseteq \mathtt{rlx}}]; (G.\mathtt{sb}; [\mathsf{F}])^?; [\mathsf{E}^{\sqsupseteq \mathtt{acq}}] \qquad (\textit{synchronizes with})$$

$$G.\mathtt{hb} \triangleq (G.\mathtt{sb} \cup G.\mathtt{sw})^+ \qquad (\textit{happens-before})$$

Happens-before is defined in terms of two more basic definitions. First, the *release sequence* ($\mathtt{rseq}$) of a write contains the write itself and all later writes to the same location in the same thread, as well as all RMWs that recursively read from such writes. Next, a release event *a synchronizes with* ($\mathtt{sw}$) an acquire event b, whenever b (or, in case b is a fence, some $\mathtt{sb}$-prior read) reads from the release sequence of a (or, in case a is a fence, of some $\mathtt{sb}$-later write). Finally, we say that an event *a happens-before* another event b if there is a path from a to b consisting of $\mathtt{sb}$ and $\mathtt{sw}$ edges.

To order accesses to a given location, the model requires that for every location x, $G.\mathtt{mo}$ *totally* orders the writes to x, and defines an extension of $\mathtt{mo}$, which is a partial order on *all* accesses to x:

$$G.\mathtt{eco} \triangleq (G.\mathtt{mo} \cup G.\mathtt{rf} \cup G.\mathtt{rf}^{-1}; G.\mathtt{mo})^+ \qquad (\textit{extended coherence order})$$

Here, writes are ordered using $\mathtt{mo}$, while reads are placed after the writes they read from, but before writes that are $\mathtt{mo}$-later than the writes they read from. Then, the coherence condition simply requires that $G.\mathtt{eco}; G.\mathtt{hb}$ is irreflexive.

Note that, assuming the $G.\mathtt{mo}$ totally orders same-location writes, an equivalent definition of the extended coherence order is given by $G.\mathtt{eco} = G.\mathtt{mo}; G.\mathtt{rf}^? \cup G.\mathtt{rf} \cup G.\mathtt{rf}^{-1}; G.\mathtt{mo}; G.\mathtt{rf}^?$.

3.2.3 Atomicity. The atomicity constraint requires that for every pair $\langle r, w\rangle \in G.\mathtt{rmw}$, there is no event in modification order between the write from which r reads-from and w. Accordingly, it requires that $G.\mathtt{rf}; G.\mathtt{rmw}; G.\mathtt{mo}^{-1}; G.\mathtt{mo}^{-1}$ is irreflexive.

In particular, this condition (together with totality of $\mathtt{mo}$) disallows two RMWs to read from the same write (i.e., $G.\mathtt{rf}; G.\mathtt{rmw}$ is partial function). Note, however, if some exclusive read is $\mathtt{sb}$-maximal it may read from a write that is read by another exclusive read (as, e.g., happens during the exploration of FAIS in §2.4). We refer to $\mathtt{sb}$-maximal exclusive reads as *pending RMWs*, because their corresponding write event has not been added to the execution yet, and denote this set by $G.\mathsf{R}^{\mathtt{ex}}_{\mathtt{pending}}$ (formally, $G.\mathsf{R}^{\mathtt{ex}}_{\mathtt{pending}} \triangleq G.\mathsf{R}^{\mathtt{ex}} \setminus dom(G.\mathtt{rmw})$).

3.2.4 Global SC constraint. SC accesses and fences are subject to a global constraint, which, roughly speaking, requires threads to agree on their order. In fact, due to the interaction with other access modes, this is notably the most involved part of RC11, which addresses flaws of the original C/C++ memory model. The repaired SC condition requires the acyclicity of a relation called *partial SC order*, denoted $\mathtt{psc}$, which is, in turn, defined using additional helper notations (we refer the reader to Lahav et al. [2017] for detailed explanations):

$$G.\mathtt{sb}|_{\neq\mathtt{loc}} \triangleq \{\langle a, b\rangle \in G.\mathtt{sb} \mid \mathtt{loc}(a) \neq \mathtt{loc}(b)\} \qquad G.\mathtt{hb}|_{\mathtt{loc}} \triangleq \{\langle a, b\rangle \in G.\mathtt{hb} \mid \mathtt{loc}(a) = \mathtt{loc}(b)\}$$

$$G.\mathtt{scb} \triangleq G.\mathtt{sb} \cup G.\mathtt{sb}|_{\neq\mathtt{loc}}; G.\mathtt{hb}; G.\mathtt{sb}|_{\neq\mathtt{loc}} \cup G.\mathtt{hb}|_{\mathtt{loc}} \cup G.\mathtt{mo} \cup G.\mathtt{rf}^{-1}; G.\mathtt{mo} \qquad (\textit{SC-before})$$

$$\begin{aligned} G.\mathtt{psc} \triangleq\ & ([\mathsf{E}^{\mathtt{sc}}] \cup [\mathsf{F}^{\mathtt{sc}}]; G.\mathtt{hb}^?); G.\mathtt{scb}; ([\mathsf{E}^{\mathtt{sc}}] \cup G.\mathtt{hb}^?; [\mathsf{F}^{\mathtt{sc}}]) \ \cup \\ & [\mathsf{F}^{\mathtt{sc}}]; (G.\mathtt{hb} \cup G.\mathtt{hb}; G.\mathtt{eco}; G.\mathtt{hb}); [\mathsf{F}^{\mathtt{sc}}] \qquad (\textit{partial SC order}) \end{aligned}$$

3.2.5 No $\mathtt{sb} \cup \mathtt{rf}$ cycles. Finally, in order to rule out "out-of-thin-air" behaviors, where reads can return arbitrary values due to cyclic dependencies, RC11 adopts a conservative fix over the original C/C++11 model suggested by Vafeiadis and Narayan [2013] and Boehm and Demsky [2014]. It requires that the relation $G.\mathtt{sbrf} \triangleq (G.\mathtt{sb} \cup G.\mathtt{rf})^+$ is irreflexive.

3.2.6 RC11 consistency. Summarizing the constraints above, we define RC11-consistency.

Definition 3.7. An execution G is RC11-*consistent* if the following hold:

- G is complete. (COMPLETENESS)
- For every location x, $G.\mathtt{mo}$ totally orders $G.\mathsf{E} \cap \mathsf{W}_x$. (VALID MO)
- $G.\mathtt{eco}; G.\mathtt{hb}$ is irreflexive. (COHERENCE)
- $G.\mathtt{rf}; G.\mathtt{rmw}; G.\mathtt{mo}^{-1}; G.\mathtt{mo}^{-1}$ is irreflexive. (ATOMICITY)
- $G.\mathtt{psc}$ is acyclic. (SC-ACYCLICITY)
- $G.\mathtt{sbrf}$ is irreflexive. (SBRF)

We will refer to executions that satisfy all conditions except (possibly) for SC-ACYCLICITY as RC11-*preconsistent* executions.

Next, we can formally state the "prefix-closedness" and "prefix-determinedness" properties mentioned earlier. For a set E such that $dom(\mathtt{sbrf}; [E]) \subseteq E$, $\mathsf{Restrict}(G, E)$ denotes the execution G' given by $G'.\mathsf{E} = E$, $G'.\mathtt{rf} = [E]; G.\mathtt{rf}; [E]$, and $G'.\mathtt{mo} = [E]; G.\mathtt{mo}; [E]$.

LEMMA 3.8 (PREFIX-CLOSED). *If G is RC11-(pre)consistent, then so is $\mathsf{Restrict}(G, E)$ for every $E \subseteq G.\mathsf{E}$ such that $dom(\mathtt{sbrf}; [E]) \subseteq E$.*

PROOF. Let $G' = \mathsf{Restrict}(G, E)$. It is easy to see that for every relation R mentioned above, we have $G'.R = [E]; G.R; [E]$. The claim follows observing that all conditions in Def. 3.7, except for

COMPLETENESS, are monotone: if they hold for larger relations, they also hold for smaller ones. Finally, completeness of G' follows from the completeness of G and the fact that $dom(\mathsf{rf}; [E]) \subseteq E$. □

Lemma 3.9 (prefix-determined). *Let r be an sb-maximal read event in an execution G. If both* $\mathrm{Restrict}(G, G.\mathsf{E} \setminus \{r\})$ *and* $\mathrm{Restrict}(G, dom(G.\mathsf{sbrf}^?; [r]))$ *are* RC11*-preconsistent, then so is G.*

Proof. Easily follows from the definitions: any violation of one of the consistency conditions (except for SC-ACYCLICITY) is a violation of the same condition either in $\mathrm{Restrict}(G, G.\mathsf{E} \setminus \{r\})$ (if the violation does not involve r) or in $\mathrm{Restrict}(G, dom(G.\mathsf{sbrf}^?; [r]))$ (if the violation involves r). For example, consider a violation of COHERENCE, i.e., a pair $\langle a, b\rangle \in G.\mathsf{hb}$ such that $\langle b, a\rangle \in G.\mathsf{eco}$. If $\langle b, r\rangle \in G.\mathsf{sbrf}^?$, then, since $G.\mathsf{hb} \subseteq G.\mathsf{sbrf}$, we have also $\langle a, r\rangle \in G.\mathsf{sbrf}$. Hence, $a, b \in dom(G.\mathsf{sbrf}^?; [r])$, and it follows that $\langle a, b\rangle \in G'.\mathsf{hb}$ and $\langle b, a\rangle \in G'.\mathsf{eco}$, where $G' = \mathrm{Restrict}(G, dom(G.\mathsf{sbrf}^?; [r]))$. Thus, COHERENCE is violated also in $\mathrm{Restrict}(G, dom(G.\mathsf{sbrf}^?; [r]))$. Alternatively, if $\langle b, r\rangle \notin G.\mathsf{sbrf}^?$, we have that $b \neq r$, as well as $a \neq r$ (since r is hb-maximal in G), and so COHERENCE is violated also in $\mathrm{Restrict}(G, G.\mathsf{E} \setminus \{r\})$. □

Note that SC is not prefix-determined (and thus, RC11-consistency is not prefix-determined due to SC-ACYCLICITY). Indeed, consider the execution of the "store buffering" program in Fig. 6, and let r be any of the two reads in this execution. The executions $\mathrm{Restrict}(G, G.\mathsf{E} \setminus \{r\})$ and $\mathrm{Restrict}(G, dom(G.\mathsf{sbrf}^?; [r]))$ are both SC-consistent, but G is not.

3.3 Semantics of concurrent programs

To complete the description of the concurrency semantics, we explain how traces of sequential programs and RC11-consistent executions interact.

First, from a given complete execution graph, one can easily extract a trace for each thread. Formally, $\mathrm{trace}(G, i) = \langle G.\mathtt{val}_{\mathtt{r}}(a_1), \dots, G.\mathtt{val}_{\mathtt{r}}(a_n)\rangle$ where $a_1, \dots, a_n$ is the enumeration of $G.\mathsf{E} \cap \mathsf{E}_i$ following sb. Then, we say that G is an *execution of* a program $P = \|_{i \in \mathsf{Tid}} P_i$ if for every thread $i \in \mathsf{Tid}$ and a proper prefix t of length n of $\mathrm{trace}(G, i)$, we have $P_i(t) = \mathtt{lab}(a)$ where a is the $n + 1$ event (following sb) in $G.\mathsf{E} \cap \mathsf{E}_i$. In turn, G is called *full* if $P_i(\mathrm{trace}(G, i)) = \bot$ for every thread $i \in \mathsf{Tid}$.

Now, we can define when a program is *erroneous*. There are two kinds of errors. The first is an assertion violation indicated by an error event in some consistent execution of the program. The second is a forbidden data race (a race that involves a non-atomic access) as defined next.

Definition 3.10. Two events $a, b \in \mathsf{E}$ are called *conflicting* if $\mathtt{W} \in \{\mathtt{typ}(a), \mathtt{typ}(b)\}$, $a \neq b$, and $\mathtt{loc}(a) = \mathtt{loc}(b)$. A pair $\langle a, b\rangle$ is called a *race* in an execution G (denoted $\langle a, b\rangle \in G.\mathsf{race}$) if a and b are conflicting, $a, b \in G.\mathsf{E}$, $\mathsf{na} \in \{\mathtt{mod}(a), \mathtt{mod}(b)\}$, and $\langle a, b\rangle \notin G.\mathsf{hb} \cup G.\mathsf{hb}^{-1}$.

Remark 1. One subtle point to note is that non-atomic reads may still be involved in a race even if they can read from only one write. For example, the only consistent execution of the program

$$x_{\mathtt{rlx}} := 1 \;\Big\|\; a := x_{\mathtt{rlx}}; \mathbf{if}\ a\ \mathbf{then}\ b := x_{\mathtt{na}}$$

containing a non-atomic read is one where the relaxed read of x reads the value 1. In this case, however, the non-atomic read cannot read 0, because that violates COHERENCE. Nevertheless, it does race with the $x_{\mathtt{rlx}} := 1$ because there is no hb between the two events. (Recall that reads-from between relaxed accesses does not contribute to happens-before).

Definition 3.11. A program P is *erroneous* if there exists an RC11-consistent execution G of P such that either $G.\mathsf{E} \cap \mathtt{error} \neq \emptyset$ or $G.\mathsf{race} \neq \emptyset$.

Algorithm 1 Main exploration algorithm.

```
procedure VISIT($G, T$)
    $a \leftarrow \mathsf{next}_P(G, T)$
    if $a \neq \bot$ then
        $G \leftarrow \mathsf{Add}(G, a)$
        switch $\mathtt{typ}(a)$ do
            case R        VISITREAD($G, T, a$)                 ▹ Handling a read event
            case W        VISITWRITE($G, T, a$)                ▹ Handling a write event
            otherwise     VISIT($G, T$)      ▹ Handling a fence, a blocked, or an error event
    else if $(G.\mathsf{E} \cap \mathtt{error} \neq \emptyset \vee G.\mathtt{race} \neq \emptyset) \wedge (G.\mathtt{psc}$ is acyclic) then
        exit(erroneous program)
```

4 OUR MODEL CHECKING ALGORITHM

In this section, we present our algorithm for model checking concurrent programs under RC11. Consider a program $P = \|_{i \in \mathsf{Tid}} P_i$, where each P_i is a sequential loop-free deterministic program. As outlined above, the algorithm maintains a collection of program executions annotated with a set of "revisitable" reads. We refer to such pairs as *configurations*:

Definition 4.1. A *configuration* (of program P) is a pair $\langle G, T \rangle$, where G is an RC11-preconsistent execution of P and $T \subseteq G.\mathsf{E} \cap \mathsf{R}$ is a set of *revisitable read events* such that the following hold:

(1) $\mathit{codom}([T]; G.\mathtt{sbrf}; [\mathsf{R}]) \subseteq T$.
(2) $[G.\mathsf{R}^{\mathsf{ex}}_{\mathtt{pending}}]; G.\mathtt{rf}^{-1}; G.\mathtt{rf}; [G.\mathsf{R}^{\mathsf{ex}} \setminus G.\mathsf{R}^{\mathsf{ex}}_{\mathtt{pending}}] \subseteq (\mathsf{E} \setminus T) \times T$.
(3) $G.\mathsf{R}^{\mathsf{ex}}_{\mathtt{pending}}$ is either empty, a singleton, or consists of two events reading from the same write, one revisitable and one not—namely, two events $\langle r, t \rangle \in [\mathsf{E} \setminus T]; G.\mathtt{rf}^{-1}; G.\mathtt{rf}; [T]$.

Our exploration algorithm maintains the conditions in Def. 4.1 by construction. The first condition observes that the revisit set is $\mathtt{sbrf}$-closed; i.e., if an event is revisitable, then so are all its $\mathtt{sbrf}$-later read events. Condition 2 requires that if the graph contains a pending exclusive read r and non-pending exclusive read t, reading from the same write, then t must be revisitable and r should not. The requirement that t is revisitable is to avoid latent atomicity violations (otherwise, atomicity would be violated by adding r's corresponding exclusive write to the execution). Finally, condition 3 ensures that pending exclusive reads are completed as soon as possible by adding their corresponding exclusive writes. (Once an execution has a pending exclusive read, the next event to add is its corresponding write.) During the revisiting of read events due to CAS instructions, however, it may also happen that two reads reading from the same write may become simultaneously pending; this will be explained in §4.4.

To decide on the order in which events are added to G, the exploration algorithm assumes the existence of a partial function, $\mathsf{next}_P : \mathsf{Configuration} \rightharpoonup \mathsf{E}$, that given a configuration $\langle G, T \rangle$ of P generates a new event to be added to G. The function next_P could be implemented by choosing a thread $i \in \mathsf{Tid}$ such that $P_i(\mathsf{trace}(G, i)) \neq \bot$, and taking $\mathsf{next}_P(G, T)$ to be $\langle i, |G.\mathsf{E} \cap \mathsf{E}_i| + 1, P_i(\mathsf{trace}(G, i)) \rangle$. For correctness, the choice of i should satisfy the following:

(1) If $G.\mathsf{R}^{\mathsf{ex}}_{\mathtt{pending}} = \{r\}$ then $i = \mathtt{tid}(r)$.
(2) Otherwise, if $G.\mathsf{R}^{\mathsf{ex}}_{\mathtt{pending}} = \{r, t\}$ where $r \notin T$ and $t \in T$, then $i = \mathtt{tid}(r)$.
(3) Otherwise, i can be chosen arbitrarily among all threads that are not blocked or finished. (In our implementation, we just select the one with the smallest identifier.)

The function next_P returns $\bot$ whenever either all threads have finished or are blocked.

Algorithm 2 Visiting a read r, which involves calculating where r could read from.

```
1:  procedure VisitRead($G, T, r$)
2:      $W \leftarrow G.\mathsf{E} \cap \mathsf{W}_{\mathtt{loc}(r)}$            ▷ Consider all the writes in $G$ from where to read
3:      $W \leftarrow W \setminus dom(G.\mathtt{mo}; G.\mathtt{rf}^?; G.\mathtt{hb}; [r])$   ▷ Remove ones that would violate COHERENCE
4:      $A_1 \leftarrow \{w \in \mathsf{W} \mid \mathtt{val}(w) \in \mathtt{exvals}(r)\}$
5:      $A_2 \leftarrow dom(G.\mathtt{rf}; ([G.\mathsf{R}^{\mathsf{ex}} \setminus T] \cup [G.\mathsf{R}^{\mathsf{ex}} \cap T]; G.\mathtt{sbrf}; [r]))$
6:      $W \leftarrow W \setminus (A_1 \cap A_2)$                  ▷ Remove ones that would violate ATOMICITY
7:      choose some $w_0 \in W \cap dom(G.\mathtt{sbrf}; [r])$
8:      while $\mathtt{val}(w_0) \in \mathtt{exvals}(r) \land codom([w_0]; G.\mathtt{rf}; G.\mathtt{rmw}) \neq \emptyset$ do
9:          $\{w_0\} \leftarrow codom([w_0]; G.\mathtt{rf}; G.\mathtt{rmw})$
10:     Visit(SetRF($G, w_0, \{r\}$), $T \cup \{r\}$)
11:     for $w \in W \setminus \{w_0\}$ do Visit(SetRF($G, w, \{r\}$), $T \setminus dom(G.\mathtt{sbrf}; [r, w])$)
```

4.1 The main routine

A pseudocode implementation of the exploration algorithm is given in Algorithm 1. The procedure Visit(G, T) explores all the configurations of the program P that are reachable from the configuration $\langle G, T\rangle$. Initially, it is called with the initial graph G_0 containing only the initialization writes E_0 and the empty revisit set $T_0 = \emptyset$.

Visit first calls $\mathtt{next}_P(G, T)$ to return the next new event to be added. If a concrete event is returned, it is added to the execution graph by putting it at the end of the appropriate thread of the graph. This is done using the following construction:

Definition 4.2 (Addition). For an event $a = \langle i, |G.\mathsf{E} \cap \mathsf{E}_i| + 1, l\rangle$, $\mathsf{Add}(G, a)$ is the execution G' given by $G'.\mathsf{E} = G.\mathsf{E} \uplus \{a\}$, $G'.\mathtt{rf} = G.\mathtt{rf}$, and $G'.\mathtt{mo} = G.\mathtt{mo}$.

Depending on whether the new event, a, is a read, a write, or some other event, Visit then calls VisitRead, VisitWrite or Visit recursively. If no event is returned, it means that we have reached a full execution. If the execution contains an error or a race and also satisfies the SC-ACYCLICITY constraint, the exploration terminates and reports an error.

Remark 2. Our implementation of Algorithm 1 improves it in a few simple ways. When an error or a racy event is added to an execution, if the execution satisfies SC-ACYCLICITY, the error is reported immediately. Similarly, when a blocked event is added and its $\mathtt{sbrf}$-prefix contains no revisitable reads and at least one non-revisitable read, further exploration is aborted. Indeed, in this case, our construction ensures that there is another execution with a revisitable read $\mathtt{sbrf}$-before the blocked event, whose exploration is not aborted.

4.2 The VisitRead procedure (Algorithm 2)

VisitRead calculates the set of writes W that the read r could possibly read from. These are the set of all writes to the same location as r that belong to the execution graph G, where reading from them violates neither COHERENCE nor ATOMICITY. Coherence is violated by reading from a write that reaches r via $\mathtt{mo}; \mathtt{rf}^?; \mathtt{hb}$ because then $\mathtt{eco}$ would contradict $\mathtt{hb}$. Atomicity is violated by reading from a write in $A_1 \cap A_2$; reading from these writes would make the read r become RMW event (cf. the set A_1), and moreover, these writes are already read by a non-revisitable RMW or by a revisitable RMW that is $G.\mathtt{sbrf}$-before r (cf. the set A_2).

Then, lines 7–9 concern choosing an appropriate default write w_0 from the set W. Reading from that write makes the read r revisitable (line 10), whereas reading from any other write w not

Algorithm 3 Visiting a write w, which involves adding it in $\mathtt{mo}$ and revisiting any relevant reads.

```
1: procedure VisitWrite($G, T, w$)
2:     if $\exists w_p.\ \langle w_p, w\rangle \in G.\mathtt{rf}; G.\mathtt{rmw}$ then
3:         RevisitReads(InsertMO($G, w_p, w$), $T$, $w$)
4:     else
5:         $w_0 \leftarrow \max_{G.\mathtt{mo}} \mathsf{W}_{\mathtt{loc}(w)}$
6:         RevisitReads(InsertMO($G, w_0, w$), $T$, $w$)
7:         for $w_p \in G.\mathsf{E} \cap \mathsf{W}_{\mathtt{loc}(w)} \setminus \{w_0\} \setminus dom(G.\mathtt{rf}; [G.\mathsf{R}^{\mathsf{ex}}] \cup G.\mathtt{mo}; G.\mathtt{rf}^?; G.\mathtt{hb}; [w])$ do
8:             RevisitReads(InsertMO($G, w_p, w$), $T \setminus dom(G.\mathtt{sbrf}; [w, \mathrm{succ}_{G.\mathtt{mo}}(w_p)])$, $w$)
```

only makes the read r non-revisitable but also makes any reads in the $\mathtt{sbrf}$-prefix of w or of r non-revisitable (line 11). Adding the new reads-from edge is done via the following construction:

Definition 4.3 ($\mathtt{rf}$-setting). For an event $w \in G.\mathsf{E} \cap \mathsf{W}$ and a set $R \subseteq G.\mathsf{E} \cap \mathsf{R}_{\mathtt{loc}(w)}$, $\mathsf{SetRF}(G, w, R)$ is the execution G' given by $G'.\mathsf{E} = G.\mathsf{E}$, $G'.\mathtt{rf} = G.\mathtt{rf} \setminus (\mathsf{E} \times R) \cup (\{w\} \times R)$, and $G'.\mathtt{mo} = G.\mathtt{mo}$.

We note that the actual choice of $w_0 \in W$ is not important for soundness, but it affects termination and our optimality result. We generally prefer to read from some $\mathtt{sbrf}$-prior event, as this is needed for optimality. If the chosen w_0 is already read by an exclusive read and reading from w_0 would also make r exclusive, we select the next event down the $\mathtt{rf}; \mathtt{rmw}$ chain. (The latter is needed to ensure termination, e.g., for the program consisting of three parallel $\mathsf{FAI}(x)$ instructions.)

4.3 The VisitWrite procedure (Algorithm 3)

We proceed to the case when a write w is added to an execution graph G. VisitWrite determines the possible places of the write in the modification order for the location it writes to, and then for each such place, it calls the RevisitReads procedure to determine all the ways in which existing reads could be revisited to read from the newly added write w. If the write is exclusive (i.e., a part of an RMW instruction), then its placement in $\mathtt{mo}$ is unique: it must immediately follow the write, w_p, from which the read event of the RMW reads (cf. lines 2–3).

Otherwise, w can be placed after the maximal write in $\mathtt{mo}$ as well as immediately after any other write w_p that is not read by an exclusive read and that is not $\mathtt{mo}$-overwritten by another write $\mathtt{rf}^?; \mathtt{hb}$ before w. To see why the latter condition is necessary consider w_p had some $\mathtt{mo}$-successor w_n that is $\mathtt{rf}^?; \mathtt{hb}$ before w. Placing w immediately after w_p would place it before w_n and would thus result in a $\mathtt{mo}; \mathtt{rf}^?; \mathtt{hb}$ cycle, thereby violating COHERENCE. In case the write is placed between some $\langle w_p, w_n\rangle$ pair in $G.\mathtt{mo}$ where $w_n = \mathrm{succ}_{G.\mathtt{mo}}(w_p)$, we additionally remove any events in the $\mathtt{sbrf}$-prefixes of w and of w_n from the revisit set.

The reason why the $\mathtt{sbrf}$-prefixes of w are removed from the revisit set is analogous to that for removing the $\mathtt{sbrf}$-prefixes of the read r in the VisitRead procedure in all but one subexplorations. Namely, revisiting some read $\mathtt{sbrf}$-before w will remove w and so its placement in the $\mathtt{mo}$-order in those revisited executions is irrelevant. It, therefore, suffices to have the $\mathtt{sbrf}$-prefix of w retain its revisitability status in one subexploration—here, we take the one where w becomes $\mathtt{mo}$-maximal.

The reason why the $\mathtt{sbrf}$-prefixes of w_n are also removed from the revisit set is similar. First note that the execution G_p where w is placed immediately after w_p and the execution G_n where w is placed immediately after w_n differ only in the relative order of w and w_n in $\mathtt{mo}$. (In particular, $G_p.\mathtt{mo} \setminus \{\langle w, w_n\rangle\} = G_n.\mathtt{mo} \setminus \{\langle w_n, w\rangle\}$.) Now consider some read before w_n is revisited, and so w_n is removed from the execution. Then, in those revisited executions, the relative placement of w and w_n in $\mathtt{mo}$ is irrelevant; so it does not matter whether these executions get generated by revisiting

Algorithm 4 Procedure for revisiting reads (for the case that w is not exclusive).

```
1: procedure RevisitReads($G, T, w$)
2:     $R \leftarrow T \cap \mathsf{R}_{\mathtt{loc}(w)}$                                    ▷ Revisitable reads of the same location
3:     $R \leftarrow R \setminus dom(G.\mathtt{sbrf};[w])$                                  ▷ Discard ones violating SBRF
4:     $R \leftarrow R \setminus codom([w];G.\mathtt{mo};G.\mathtt{rf}^?;G.\mathtt{hb}^?;G.\mathtt{sb})$       ▷ Discard ones violating COHERENCE
5:     for $K \subseteq R$ such that $[K];G.\mathtt{sbrf};[K] = \emptyset \;\land\; |\{r \in K \mid \mathtt{val}(w) \in \mathtt{exvals}(r)\}| \leq 1$ do
6:         $G' \leftarrow \mathsf{Remove}(G, codom([K];G.\mathtt{sbrf}))$                      ▷ Remove sbrf-successors of $K$
7:         $G' \leftarrow \mathsf{SetRF}(G', w, K)$                                         ▷ Make the $K$ reads read from $w$
8:         $T' \leftarrow (T \cap G'.\mathsf{E}) \setminus dom(G'.\mathtt{sbrf}^?;[K])$             ▷ Adjust the set of revisitable reads
9:         Visit($G', T'$)
```

G_p or G_n. Therefore, to avoid duplication, we pick G_n as the preferred execution, and so remove the $\mathtt{sbrf}$-prefixes of w_n from G_p.

Finally, the adjustment of $G.\mathtt{mo}$ is done using the following construction.

Definition 4.4 ($\mathtt{mo}$-placement). For an event $w \in G.\mathsf{E} \cap \mathsf{W} \setminus (dom(G.\mathtt{mo}) \cup codom(G.\mathtt{mo}))$ and event $w_p \in G.\mathsf{E} \cap \mathsf{W}_{\mathtt{loc}(w)}$, $\mathsf{InsertMO}(G, w_p, w)$ is the execution G' given by $G'.\mathsf{E} = G.\mathsf{E}$, $G'.\mathtt{rf} = G.\mathtt{rf}$, and $G'.\mathtt{mo} = G.\mathtt{mo} \cup (dom(G.\mathtt{mo}^?;[w_p]) \times \{w\}) \cup (\{w\} \times codom([w_p];G.\mathtt{mo}))$.

Finally, for all the valid $\mathtt{mo}$-placements of the write w, VisitWrite calls the RevisitReads procedure to consider which reads may and/or should be revisited.

4.4 The RevisitReads procedure (Algorithms 4 and 5)

The RevisitReads procedure is the most involved part of our algorithm. We present its pseudocode in Algorithm 4, first for the easier case when the freshly added write is not exclusive.

First, RevisitReads calculates the set of reads that may be revisited to read from the freshly added write w. We start with all the revisitable reads to the same location as w (line 2) and remove ones which if they were to read from w would result in a $\mathtt{sbrf}$-cycle thereby violating SBRF (line 3) as well as those which if they were to read from w would result in a COHERENCE violation (line 4).

Then, we consider revisiting every subset K of the relevant revisitable reads. To ensure that the revisits do not introduce any latent ATOMICITY violations, at line 5, we require that at most one read in K will become part of a RMW by the change. Note that when a set of reads K is revisited, the values returned by those reads may change; so all the $\mathtt{sbrf}$-successors of K have to be removed from the execution. Accordingly, line 6 creates a copy, G', of the execution containing all the events that are not $\mathtt{sbrf}$-after K, using the following construction:

Definition 4.5 (Removal). For a set $E' \subseteq G.\mathsf{E}$ such that $codom([E'];G.\mathtt{sbrf}) \subseteq E'$, we denote by $\mathsf{Remove}(G, E')$ the execution G' given by $G'.\mathsf{E} = G.\mathsf{E} \setminus E'$, $G'.\mathtt{rf} = [G'.\mathsf{E}];G.\mathtt{rf};[G'.\mathsf{E}]$, and $G'.\mathtt{mo} = [G'.\mathsf{E}];G.\mathtt{mo};[G'.\mathsf{E}]$.

Then, line 7 changes the $\mathtt{rf}$-edges of K to read from the new write w (see Def. 4.3). Line 8 adjusts the revisit set by removing all reads in the $\mathtt{sbrf}$-prefix of K, and then the subexecution is further explored.

The case where w *is* exclusive (i.e., $w \in codom(G.\mathtt{rmw})$) requires more careful attention. Its pseudocode is given in Algorithm 5 (which, in fact, generalizes Algorithm 4). In addition to the description above, this procedure checks whether there exist any writes K_{must} that *must* be revisited, because otherwise ATOMICITY would be violated (line 5). Note that if w is not exclusive, then $K_{\text{must}} = \emptyset$. If, however, w together with some read r form an RMW, then K_{must} contains the (at most one) exclusive read other than r that reads from the same write as r does, if such a read exists.

Algorithm 5 Procedure for revisiting reads after a write is added (general case).

```
1:  procedure RevisitReads($G, T, w$)
2:      $R \leftarrow T \cap \mathsf{R}_{\mathtt{loc}(w)}$                              ▷ Revisitable reads of the same location
3:      $R \leftarrow R \setminus dom(G.\mathtt{sbrf};[w])$                          ▷ Discard ones violating SBRF
4:      $R \leftarrow R \setminus codom([w];G.\mathtt{mo};G.\mathtt{rf}^?;G.\mathtt{hb}^?;G.\mathtt{sb})$      ▷ Discard ones violating COHERENCE
5:      $K_{\text{must}} \leftarrow dom([G.\mathsf{R}^{\text{ex}}];G.\mathtt{rf}^{-1};G.\mathtt{rf};G.\mathtt{rmw};[w] \setminus G.\mathtt{rmw})$   ▷ $K_{\text{must}}$ must be revisited
6:      $R \leftarrow R \setminus codom([K_{\text{must}}];G.\mathtt{sbrf}^?)$         ▷ The remaining ones, whose revisit is optional
7:      for $K_1 \subseteq R$ such that $[K_1];G.\mathtt{sbrf};[K_1] = \emptyset \ \wedge \ |\{r \in K_1 \mid \mathtt{val}(w) \in \mathtt{exvals}(r)\}| \leq 1$ do
8:          $K \leftarrow K_1 \cup (K_{\text{must}} \setminus codom([K_1];G.\mathtt{sbrf}))$      ▷ The set of reads to be revisited
9:          $G' \leftarrow \text{Remove}(G, codom([K];G.\mathtt{sbrf}))$             ▷ Remove sbrf-successors of $K$
10:         $G' \leftarrow \text{SetRF}(G', w, K)$                              ▷ Make the $K$ reads read from $w$
11:         $T' \leftarrow (T \cap G'.\mathsf{E}) \setminus dom(G'.\mathtt{sbrf}^?;[K_1])$      ▷ Adjust the set of revisitable reads
12:         Visit($G', T'$)
```

$\mathsf{CAS}(x,1,2) \parallel \mathsf{FAI}(x) \parallel \mathsf{FAI}(x)$

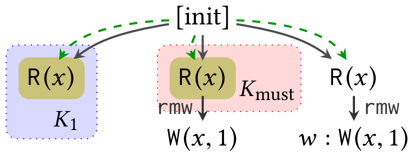

Fig. 8. Example demonstrating that two revisited reads may become exclusive by RevisitReads.

Then, intuitively, we have to consider for revisiting only subsets K that contain K_{must}. To achieve this without duplication, the process is a bit more complicated. First, at line 6, we remove K_{must} and all its sbrf-successors from R. Now R contains all the reads, whose revisit is entirely optional, while K_{must} contains those that must be revisited. Then, for every subset K_1 of R that does not reach itself with sbrf (line 7), we take the set of reads to be revisited, K, to be the union of K_1 and K_{must} (line 8). More precisely, K_{must} might be sbrf-after K_1, in which case we just take $K = K_1$.

Then, for each such construction of K_1 and K, the actual revisit takes place. As before, we create a copy, G', of the execution containing all the events that are not sbrf-after K (line 9); change the rf-edges of K (line 10); adjust the revisit set (line 11); and finally explore the subexecution.

Note that line 11 adjusts the revisit set by removing reads in the sbrf-prefix of K_1, rather than of K. (Recall Fig. 4, where we the read of thread I, though revisited, is left in the revisit set in execution ②.) The reason why the sbrf-prefix of only K_1 (and not also of K_{must}) is removed from the revisit set is that revisiting K_1 is optional, whereas revisiting K_{must} is forced. For every $r \in T \cap dom(G'.\mathtt{sbrf}^?;[K_1])$, there is a loop iteration that keeps r in the revisit set and generates a graph G'' that is equal to G' when cut right after r; in particular, note that the loop iteration with $K_1 = \emptyset$ keeps the revisit set unchanged. In contrast, K_{must} is revisited (or deleted) by all loop iterations; so if one were to remove its $\mathtt{sbrf}^?$-predecessors from the revisit set, one would fail to cover the executions where those reads need to be revisited.

Note that it is possible for both a read in K_1 and in K_{must} to become exclusive. This, for instance, occurs in the program shown in Fig. 8 when the increment of the third thread is added and reads from the initialization write. Then K_{must} includes the increment of thread II, and a possible choice for K_1 is the read of the failed CAS. Revisiting that read to read from w will make it exclusive, which will conflict with the revisited increment which will also be an exclusive read reading from w. This, however, is not a problem. The next step of the Visit procedure will add the write event of the CAS, which will cause the increment of thread II to be revisited (as it will be again in K_{must}), and a consistent subexecution will be generated, where the increment of thread II reads from the successful CAS. Note that if we instead had forbidden for K to contain two reads that would become exclusive when reading from w, the algorithm would not have been able to generate this execution and would, therefore, have been unsound.

4.5 Correctness of the algorithm

In this section, we establish the correctness of the algorithm.

Theorem 4.6. *P is erroneous iff* $\textsc{Visit}(G_0, T_0)$ *reports an error.*

The proof of this theorem requires the following additional definitions and lemmas. First, we establish an invariant of the algorithm.

Definition 4.7. Given a configuration $\langle G, T\rangle$ and a set $\mathcal{V}$, we write $\langle G, T\rangle \rightsquigarrow \mathcal{V}$ if calling $\textsc{Visit}(G, T)$ generates the immediate calls $\textsc{Visit}(G', T')$ precisely for all $\langle G', T'\rangle \in \mathcal{V}$.

Lemma 4.8. *If $\langle G, T\rangle$ is a configuration and $\langle G, T\rangle \rightsquigarrow \mathcal{V}$, then every $\langle G', T'\rangle \in \mathcal{V}$ is a configuration.*

The completeness direction (i.e., "no false positives") directly follows from this invariant since in a configuration $\langle G, T\rangle$ (of program P), we have by definition that G is an RC11-preconsistent execution of P (see Def. 4.1). Second, to guarantee termination the following lemma provides a progress measure.

Lemma 4.9. *Let $\langle G_1, T_1\rangle$ be a configuration. Suppose that $\langle G_1, T_1\rangle \rightsquigarrow \mathcal{V}$, and let $\langle G_2, T_2\rangle \in \mathcal{V}$. Then, $|\langle G_1, T_1\rangle| <_{lex} |\langle G_2, T_2\rangle|$, where $|\langle G, T\rangle| \triangleq \langle |G.\mathsf{E} \setminus codom([T]; G.\mathsf{sbrf}^?)|, |G.\mathsf{E}|\rangle$.*

The assumption that each P_i is loop-free implies that there exists a bound L such that $P_i(t) = \bot$ for every $i \in \mathsf{Tid}$ and trace t of length greater than L. Hence, for every execution G of some program with N threads, we have $|G.\mathsf{E} \setminus \mathsf{E}_0| \leq N \times L$. It follows that, for a given program, $|\langle G, T\rangle|$ is bounded, and so the termination of the algorithm is guaranteed.

Third, the soundness direction of the proof (i.e., "all errors are reported") requires us to show that all RC11-preconsistent executions of P are 'covered' by the exploration algorithm. This is proved inductively: the initial configuration trivially covers all RC11-preconsistent executions of P, and every execution covered by $\langle G, T\rangle$ is guaranteed to be covered by some $\langle G', T'\rangle$ that is generated by calling $\textsc{Visit}(G, T)$. Formally, 'covering' is defined as follows:

Definition 4.10. For an execution G and a set M of read events, $\mathsf{Cover}(G, M)$ consists of all RC11-preconsistent executions G_f of P for which the following hold, where $E = G.\mathsf{E} \setminus codom([M]; \mathsf{sb}^?)$:

- $E \cup M \subseteq G_f.\mathsf{E}$
- $G.\mathsf{rf}; [E] \subseteq G_f.\mathsf{rf}$
- $[E]; G_f.\mathsf{rf}; [M] = \emptyset$
- $[E]; G.\mathsf{mo}; [E] \subseteq G_f.\mathsf{mo}$

The set of executions *covered by* a configuration $\langle G, T\rangle$, denoted $[\![\langle G, T\rangle]\!]$ is given by

$$[\![\langle G, T\rangle]\!] \triangleq \bigcup \{\mathsf{Cover}(G, M) \mid M \subseteq T, codom([M]; G.\mathsf{sbrf}) \subseteq codom([M]; \mathsf{sb}^?)\}.$$

In turn, for a set $\mathcal{V}$ of configurations, we take $[\![\mathcal{V}]\!] \triangleq \bigcup \{[\![\langle G, T\rangle]\!] \mid \langle G, T\rangle \in \mathcal{V}\}$.

Intuitively, in the definition of $\mathsf{Cover}(G, M)$, the set M corresponds to a set of read events whose revisiting will generate G_f. Thus, $G_f \in \mathsf{Cover}(G, M)$ if removing all events $G.\mathsf{sbrf}$-after M from G yields an execution that is a prefix of G_f in which the reads in M are unresolved (do not read from any write). In particular, note that G itself is always covered by $\langle G, T\rangle$ (pick $M = \emptyset$). In turn, a configuration $\langle G, T\rangle$ covers all executions G_f that are in $\mathsf{Cover}(G, M)$ for some minimal set $M \subseteq T$. For example, let $\langle G, T\rangle$ be the configuration ⓪ depicted in Fig. 4. Then, it covers the two other executions in the figure: execution ① is covered by taking $M = \emptyset$, while execution ② is covered by taking M to be the unique revisitable event in G.

The inductive step is given in the following lemma.

Lemma 4.11. *For every configuration $\langle G, T\rangle$, if $\langle G, T\rangle \leadsto \mathcal{V}$ and $\mathcal{V} \neq \emptyset$, then $[\![\langle G, T\rangle]\!] \subseteq [\![\mathcal{V}]\!]$.*

Finally, when no further calls are performed, the only execution covered is G itself.

Lemma 4.12. *For every configuration $\langle G, T\rangle$, if $\langle G, T\rangle \leadsto \emptyset$, then $[\![\langle G, T\rangle]\!] = \{G\}$.*

4.6 Optimality in the absence of RMWs and SC atomics

We move on to optimality. A run of the algorithm generates a tree of configurations starting from the initial configuration $\langle G_0, T_0\rangle$ and following the $\leadsto$ relation. Optimality means that (1) all terminal configurations are RC11-consistent executions of P and (2) for every two distinct nodes in this tree $\langle G, T\rangle$ and $\langle G', T'\rangle$, we have $G \neq G'$.

For the former, it suffices for the program to contain no SC atomics, as then $G.\mathtt{psc}$ is trivially acyclic for every reachable configuration $\langle G, T\rangle$, and so RC11-consistency of G coincides with its RC11-preconsistency. To establish the latter property, we first show that immediate sibling configurations (namely, those generated by the same parent configuration) have disjoint cover sets.

Lemma 4.13. *If $\langle G, T\rangle$ is a configuration and $\langle G, T\rangle \leadsto \mathcal{V}$, then $[\![\langle G_1, T_1\rangle]\!] \cap [\![\langle G_2, T_2\rangle]\!] = \emptyset$ for every pair $\langle G_1, T_1\rangle, \langle G_2, T_2\rangle$ of distinct configurations in $\mathcal{V}$.*

Lemma 4.13 does not suffice to prove our statement because, e.g., it does not preclude the case where the cover of a configuration intersects that of a sibling of its parent. In fact, this may be the case in the presence of RMWs (see §2.4). For example, consider the fais program (Fig. 4) augmented with a third thread writing $x := 42$, and let G_f be the full execution of the program in which the first thread reads 42 and the second reads 0. Then, G_f is not covered by configuration ① in Fig. 4, and yet it is covered both by configuration ② and by the configuration generated from ① after adding the $x := 42$ write and revisiting the read in the first thread.

However, in the absence of RMWs, VisitRead always picks w_0 to be $G.\mathtt{sbrf}$-before the freshly added read r, and so all reachable configurations are *simple*, as defined below:

Definition 4.14. A configuration $\langle G, T\rangle$ is called *simple* if $[T]; G.\mathtt{sbrf} \subseteq \mathtt{sb}$.

We further show that children of simple configurations cover only executions covered by their parents (unlike configuration ① in Fig. 4, which is not simple).

Lemma 4.15. *If $\langle G, T\rangle$ is a simple configuration and $\langle G, T\rangle \leadsto \mathcal{V}$, then $[\![\mathcal{V}]\!] \subseteq [\![\langle G, T\rangle]\!]$.*

Putting these two lemmas together, by induction, we can deduce that distinct nodes in the configuration tree cover different executions, which leads to our optimality theorem.

Theorem 4.16 (Optimality). *Let P be a program containing no SC atomics and no RMWs. Then:*

- *G is RC11-consistent in every configuration $\langle G, T\rangle$ of P.*
- *$G_1 \neq G_2$ for every two distinct calls Visit(G_1, T_1) and Visit(G_2, T_2) that are (recursively) generated by calling Visit(G_0, T_0).*

5 ADAPTATION TO THE WEAK RC11 MODEL

In this section, we present the WRC11 (Weak RC11) model—the weakening of the RC11 model briefly mentioned in §2.3. The motivation for WRC11 arises from our observation that unordered concurrent writes to the same location seldom appear in real programs. Nevertheless, a significant part of RC11—as well as of almost any other memory model we know—is dedicated to handle such cases. Indeed, the whole purpose of the $\texttt{mo}$ relation is to totally order the –otherwise unordered– writes to the same location. This total order is then used to ensure the coherence property, which, roughly speaking, enforces the reads of each thread to respect $\texttt{mo}$. When one is unconcerned about concurrent writes, there is no need to ensure this property for them. As a result, we are able to greatly simplify program execution graphs, and, consequently, obtain a simplified version of our exploration algorithm.

Formally, WRC11-executions are defined just like executions above, but they do not include the $\texttt{mo}$ component. In turn, WRC11-(pre)consistency is defined exactly as RC11-(pre)consistency except that:

- the VALID MO axiom is dropped; and
- all other mentions of $G.\texttt{mo}$ in the consistency predicates are replaced by

$$G.\texttt{mo}_{\text{weak}} \triangleq \bigcup_{x \in \text{Loc}} [\texttt{W}_x]; (G.\texttt{hb} \cup G.\texttt{rf}_x)^+; [\texttt{W}_x],$$

where $G.\texttt{rf}_x \triangleq \{\langle w, r\rangle \in G.\texttt{rf} \mid \texttt{loc}(w) = \texttt{loc}(r) = x\}$.

Intuitively, the resulting model ensures that each thread's reads respect $\texttt{mo}_{\text{weak}}$. This means that if a thread reads from some write, all subsequent operations of this thread will not read from an $\texttt{mo}_{\text{weak}}$-earlier write. However, it may read in any order from $\texttt{mo}_{\text{weak}}$-unordered writes, and may also "oscillate" between their values. This is similar to what is guaranteed by a weak form of causal consistency, studied, e.g., by Bouajjani et al. [2017]. For example, consider the following program:

$$x_{\texttt{rlx}} := 1 \;\Big\|\; x_{\texttt{rlx}} := 2 \;\Big\|\; \begin{array}{l} a := x_{\texttt{rlx}}; \\ b := x_{\texttt{rlx}}; \\ c := x_{\texttt{rlx}} \end{array} \tag{WW3R}$$

Under RC11, if $a = 1$ and $b = 2$, then we know that $c = 2$ (to obtain this outcome, $\texttt{mo}$ orders the $x := 1$ write before $x := 2$). Since the two writes are unordered by $\texttt{mo}_{\text{weak}}$ (in any execution of this program), WRC11-consistency does not ensure this property and allows the outcome $a = c = 1$ while $b = 2$. Note that reading $b = 0$ (the initial value) after $a = 1$ or $a = 2$ is not allowed in WRC11, since the initialization write is $\texttt{hb}$-before—and thus $\texttt{mo}_{\text{weak}}$-before—any other write.

To see the benefit of including $\texttt{rf}_x$ in the definition of $\texttt{mo}_{\text{weak}}$, consider the following example where FAI returns the value it read before the increment:

$$a := \text{FAI}_{\texttt{rlx}}(x) \;\Big\|\; \begin{array}{l} b := \text{FAI}_{\texttt{rlx}}(x); \\ c := x_{\texttt{rlx}} \end{array}$$

Under RC11, if $a = 0$ and $b = 1$, then the only allowed value for c is 2 (to obtain this outcome, $\texttt{mo}$ orders the write of the first thread before the write of the second). This is also the case for WRC11, since $a = 0$ and $b = 1$ entails a reads-from edge from the write of the first thread to the read of the second, resulting in $\texttt{mo}_{\text{weak}}$ from the write of the first thread to the write of the second. Had we not included $\texttt{rf}_x$ in $\texttt{mo}_{\text{weak}}$, our weakened model would have allowed the outcome $a = 0$ and $b = c = 1$. We note that the "linuxrwlocks" program mentioned in §7 contains a similar pattern.

It is easy to see that VALID MO and COHERENCE ensure that $G.\texttt{mo}_{\text{weak}} \subseteq G.\texttt{mo}$ in RC11-preconsistent executions, and therefore that WRC11-(pre)consistency is indeed weaker than RC11-(pre)consistency. In fact, our definition of $G.\texttt{mo}_{\text{weak}}$ is the largest possible relation (defined only in terms of program order and reads-from) that is contained in $G.\texttt{mo}$. Furthermore, if there are no write-write races (or, more generally, if all writes to each location x are totally ordered by $(G.\texttt{hb} \cup G.\texttt{rf}_x)^+$), then $G.\texttt{mo}_{\text{weak}} = G.\texttt{mo}$, and so WRC11-(pre)consistency and RC11-(pre)consistency coincide.

Adapting the exploration algorithm to WRC11 is straightforward:

- All mentions of $\mathtt{mo}$ are replaced with $\mathtt{mo}_{\text{weak}}$. This, in particular, makes line 4 of RevisitReads unnecessary, as w is maximal in $\mathtt{hb} \cup \mathtt{rf}$.
- VisitWrite(G, T, w) is changed to just call RevisitReads(G, T, w) *without any other work.*

Comparing the two approaches, recording $\mathtt{mo}$ explicitly ensures that only coherent executions are explored, but it comes at a certain cost: the exploration algorithm has to consider all the possible ways that concurrent writes to the same location could be totally ordered even if these writes are never read. In contrast, under WRC11, there is no need to maintain $\mathtt{mo}$, but, in some cases, many incoherent executions will be unnecessarily considered. One can easily construct toy examples showing the potential advantage of each approach, specifically ones where the unnecessary enumeration of $\mathtt{mo}$ dominates the verification time or showing that the exploration of incoherent executions is extremely costly:

- For a program consisting solely of N concurrent writes to the same location we will explore $N!$ executions in RC11, but all of them have the same behavior (and expose similar "bugs"). Under WRC11, this program has just one consistent execution. The "casw" benchmark in §7 is similar.
- For a variant of the ww3R program above with N reads instead of three, we will have a quadratic number of consistent executions under RC11, while, under WRC11, we will explore exponential (in N) number of executions.

In the vast majority of programs we examined, including all the real-world programs mentioned in §7, $\mathtt{mo}_{\text{weak}}$-unordered writes to the same location do not appear, and the difference between the run time of the two algorithms is practically negligible.

Remark 3. Separation-logic-based program logics for the release/acquire fragment of the C/C++11 model (which is identical to the release/acquire fragment of RC11) are essentially making a similar simplification, and do not support reasoning about coherence for concurrent writes. In particular, RSL [Vafeiadis and Narayan 2013] is sound for a similar weakening of its underlying model obtained by replacing $\mathtt{mo}$ by $\bigcup_{x \in \mathsf{Loc}} [\mathtt{W}_x]; G.\mathtt{hb}; [\mathtt{W}_x]$ (Marko Doko, personal communication). We conjecture that the same holds for GPS [Turon et al. 2014] and its iGPS variant [Kaiser et al. 2017], and note that it does not hold for OGRA [Lahav and Vafeiadis 2015], an Owicki-Gries-style logic, which is able to reason about coherence of concurrent writes.

6 IMPLEMENTATION

In this section, we discuss the implementation of our model checking algorithm described in §4. Actually, we have two independent implementations, one written in OCaml and one in C++. Having two implementations helped the development quite a bit: we frequently compared their outcomes as a means of debugging.

The OCaml implementation was developed first and was meant mainly for quick experimentation with variants of the algorithm, which allowed us to find a number of errors in earlier versions. The implementation closely follows the recursive structure of the algorithm, but also includes a few lower-level optimizations to avoid copying the entire execution graph when unnecessary. For instance, when visiting the subexecutions generated by VisitRead, rather than copying G and suitably restricting it, the implementation updates G in place. Then, at each iteration of the loop at line 11, the implementation cuts the current graph to the set of events that were added to the graph before the read, thereby getting the exact same G that Algorithm 2 has. Copying the graph, however, cannot be fully eliminated, and is kept in the RevisitReads procedure.

The C++ tool, called RCMC, works at the level of LLVM's intermediate representation (LLVM IR). It takes as input C/C++ programs, and uses `clang` to translate them to LLVM IR. This translation saves us a lot of work, in that the thread interpreter does not need to understand all the corner cases of C/C++, but rather works with a much smaller and more regular compiler intermediate language. One downside of this approach is that the compilation of C/C++ programs to LLVM IR programs is unverified. Moreover, during this conversion, `clang` may justifiably remove some program behaviors (and therefore also program bugs). A counterargument may be that any attempt to write from scratch a tool targeting directly C/C++ will most likely also be wrong in various corner cases. Relying on the LLVM compiler infrastructure at least ensures that RCMC will encounter exactly the same bugs in the compiler frontend as the executable generated by a compiler such as `clang`. In addition, operating on the LLVM IR level renders our approach language-agnostic, which means that it can work on any language equipped with a compiler that has an LLVM frontend.

RCMC can detect assertion violations, racy non-atomic accesses, and some memory errors. When an error is detected, a full trace corresponding to the instructions of the actual source code is shown to the user. The tool also supports thread-local storage, some standard `libc` functions, as well as external function calls. Spin loops with no side-effects are transformed automatically to `assume` statements (see §2.6) and for infinite programs an unrolling option is provided. As in the algorithm described in §4, the exploration is driven by the next_P function, and the actual interpretation of the LLVM IR code is based on the interpreter `lli`, which is distributed with LLVM. In contrast to the algorithm, however, the implementation does not perform recursive calls in every occasion; a stack is used where possible in order to make the exploration faster.

In addition to the optimizations of the OCaml version, when a set of reads R has to be revisited, RCMC does not calculate the powerset of R. Instead, we perform this calculation recursively: when a read is added to a subset of R, at the next recursive calls, we will not consider reads from the same thread, as for two such reads one will be always `sbrf`-after the other. Similarly, when a read that will be exclusive is added to a subset, all other potentially exclusive reads are discarded as well.

RCMC, implements both RC11 and WRC11 versions of the algorithm, and the choice between them is controlled by a command-line switch. The tool supports both the `pthreads` and the C11's `threads` library. The versions of LLVM currently supported by RCMC are 3.5.x, 3.6.x and 3.8.x.

7 EVALUATION

In this section, we report experimental results that compare the performance of RCMC with two other stateless model checking tools, namely CDSChecker and Nidhugg. Executables for the three tools as well as all benchmark programs we used are included in the paper's artifact.

CDSChecker [Norris and Demsky 2016] is a model checker for programs written under the C/C++11 memory model of Batty et al. [2011] but it imposes its own condition for ruling out "out-of-thin-air" behaviors. CDSChecker implements some backtracking techniques for controlled repetition of some executions until all allowed program behaviors are explored. However, it can explore both *redundant* (i.e., unnecessary) and *infeasible* (i.e., prohibited by the memory model) explorations, and, although it does use some techniques and optimizations to reduce those partial explorations, these explorations can still consume much time. CDSChecker bounds the state space either by using a CHESS yield-based fairness approach [Musuvathi et al. 2008], or by using an explicit-bound fairness approach, imposed by the scheduler. Lastly, it provides a switch to handle memory liveness, which is required for programs that rely on it for termination.

Nidhugg [Abdulla et al. 2015] is a stateless model checker for C/C++ programs that use `pthreads`, which incorporates extensions for checking the effects of weak memory models employed by modern microprocessors, including TSO, PSO and POWER. The version of Nidhugg we used (0.2) employs a very effective—albeit not optimal—DPOR algorithm called source-DPOR [Abdulla et al. 2017] for SC, TSO and PSO, and a mixture of an operational and a declarative approach for POWER [Abdulla et al. 2016].

Table 1. A benchmark set where all three tools explore the optimal number of executions ("*Traces*" column). All other columns, labeled by tools and modes of their use, show times in seconds.

	Traces	CDSChecker	Nidhugg–SC	Nidhugg–TSO	Nidhugg–PSO	Nidhugg–POWER	RCMC–RC11	RCMC–WRC11
casrot(4)	14	0.00	0.10	0.10	0.11		0.04	0.04
casrot(6)	144	0.02	0.16	0.17	0.15		0.05	0.05
casrot(8)	2048	0.24	1.10	1.21	1.55		0.15	0.15
casrot(10)	38 486	5.05	23.26	28.92	33.37		2.13	2.14
readers(3)	8	0.01	0.10	0.10	0.11	0.61	0.04	0.05
readers(8)	256	0.05	0.26	0.29	0.34	193.70	0.04	0.05
readers(13)	8192	2.14	7.25	8.15	11.18	24 801.81	0.40	0.39
readers(18)	262 144	128.91	350.48	422.10	444.34	2 088 816.45	12.83	12.72

casrot(N): There are N threads. Thread $i \in [1..N]$ performs $\mathrm{CAS}_{\mathtt{rlx}}(x, i-1, i)$.
readers(N): One thread does $x_{\mathtt{rel}} := 42$ and N other threads do an acquire read from x. Taken from Abdulla et al. [2017].

We concentrate on these two tools for the following reasons: (1) CDSChecker is the only other model checker for C/C++11 we know of and thus the tool which is most similar to RCMC, and (2) Nidhugg implements a state-of-the-art DPOR algorithm and, although it does not support the C/C++11 memory model, it has been shown to outperform other model checking tools (CBMC and `goto-instrument`) for SC, TSO, and PSO. In addition, like RCMC, Nidhugg also works at the level of LLVM IR, and we have used the same LLVM version (3.8.1) for both tools. That said, despite their similarity, we stress that these tools perform model checking on memory models with *different characteristics*, so the performance comparison between them does *not* necessarily extend beyond the particular benchmarks we used.

Though none of the tools currently runs its algorithm in parallel, the machine we used is a Dell server with four Intel(R) Xeon(R) E5-4650 CPUs (2.70GHz), eight cores each (i.e., 32 physical cores in total), has 128GB of RAM and ran Debian 4.9.30-2+deb9u2. For CDSChecker, we used the arguments arguments `-m 2 -y` for all benchmarks. The first controls the liveness of the memory system, and the second enables CHESS yield-based fairness. The latter required manually adding `thrd_yield()` statements in some of the benchmarks, where appropriate. For RCMC and Nidhugg we used exactly the same benchmarks and, for infinite programs, we used the provided unroll switch. Note that Nidhugg ignores the C11 access modes, and treats all accesses as SC/TSO/PSO depending on the command line argument that selects the memory model to use. For Nidhugg-POWER, we have inserted fences appropriately in order to map the C11 primitives into the POWER instruction set. However, since Nidhugg-POWER does not support RMWs, for tables that contain benchmarks that use RMWs (casrot in Table 1 and those in Tables 2, 3 and 6), the respective column is missing.

Before presenting the benchmark results, we mention in passing that, as a sanity check, we also used many small litmus tests to check the number of consistent executions that the tools explore and compared them against those that the `herd` tool [Alglave et al. 2014] produces. At least on these litmus tests, all tools gave the expected results, but for the larger ones `herd` was significantly slower. Hence, we do not include `herd` in our comparison.

Let us now examine the results on the first two benchmarks; cf. Table 1. On these two programs, all three tools explore the same number of executions (hence we show only one "Traces" column), which in fact are the optimal ones (i.e., no tool performs any redundant exploration). Due to this reason, in these simple benchmarks, all tools in most models (SC, TSO, PSO, RC11 and WRC11) scale similarly. We remark that CDSChecker is faster when the number of executions is very small because it uses a pre-compiled file, while both Nidhugg's and RCMC's times include the time to invoke the `clang` compiler. (All times we report are in seconds.) With more explorations, RCMC is the fastest tool. In this table, the Nidhugg-POWER column sticks out, since the entries for the readers benchmark reveal that the algorithm that Nidhugg implements for POWER [Abdulla et al. 2016] is not scalable for this benchmark. On Nidhugg-POWER, readers(18) requires more than 24 days to finish. In contrast, RCMC explores the same number of traces in just a few seconds!

Table 2. Results of some benchmarks with relaxed RMWs where CDSChecker examines many infeasible executions. In contrast, Nidhugg and RCMC are optimal on ainc and indexer, while on binc they encounter sleep-set blocked and duplicate executions. Columns SC, TSO, PSO, RC11 and WRC11 show time in seconds.

	CDSChecker				Nidhugg						RCMC				
	Traces	+	*Infeasible*	*Time*	*Traces*	+	*SSB*	SC	TSO	PSO	*Traces*	+	*Duplicate*	RC11	WRC11
ainc(3)	6	+	102	0.01	6	+	0	0.11	0.10	0.11	6	+	0	0.05	0.05
ainc(4)	24	+	4013	0.24	24	+	0	0.11	0.11	0.11	24	+	0	0.05	0.05
ainc(5)	120	+	339 479	21.60	120	+	0	0.13	0.13	0.14	120	+	0	0.05	0.05
ainc(6)	720	+	65 544 463	4885.12	720	+	0	0.27	0.29	0.35	720	+	0	0.07	0.07
binc(3)	36	+	3258	0.21	36	+	0	0.10	0.10	0.10	36	+	2	0.04	0.04
binc(4)	630	+	1 486 882	100.97	576	+	0	0.21	0.23	0.29	576	+	98	0.06	0.06
binc(5)					14 400	+	240	3.90	4.60	6.30	14 400	+	5002	0.89	0.84
binc(6)					518 400	+	23 448	167.68	196.83	242.40	518 400	+	309 394	45.02	44.93
indexer(12)	8	+	1189	0.74	8	+	0	0.12	0.13	0.14	8	+	0	0.05	0.06
indexer(13)	64	+	154 562	111.29	64	+	0	0.29	0.31	0.38	64	+	0	0.10	0.10
indexer(14)	512	+	43 399 204	36 182.04	512	+	0	1.56	1.79	2.35	512	+	0	0.56	0.54
indexer(15)					4096	+	0	12.74	14.55	18.05	4096	+	0	3.76	3.75

ainc(N): There are N threads performing $\mathsf{FAI}_{\mathtt{rlx}}(x)$.
binc(N): There are N threads performing $\mathsf{FAI}_{\mathtt{rlx}}(x)$; $\mathsf{FAI}_{\mathtt{rlx}}(y)$.
indexer(N): This classic benchmark from Flanagan and Godefroid [2005] showcases races that are hard to identify statically. There are N threads, each adding four entries into a shared hash table. If a collision occurs, the next available entry in the table is used. The benchmark is designed in a way that collisions only occur for $N \geq 12$.

Table 3. Results from a benchmark where the three tools show different behavior.

	CDSChecker			RCMC–WRC11			Nidhugg–SC			Nidhugg–TSO			Nidhugg–PSO			RCMC–RC11
	Traces	*+Infeasible*	*Time*	*Traces*	*Time*	*Traces*	+	*SSB*	*Time*	+	*SSB*	*Time*	+	*SSB*	*Time*	*Time*
casw(3)	24 +	122	0.02	24	0.05	66	+	0	0.12	+	0	0.12	+	2	0.12	0.05
casw(4)	200 +	2029	0.16	200	0.05	1200	+	40	0.46	+	98	0.55	+	237	0.80	0.09
casw(5)	2160 +	36 240	2.84	2160	0.14	32 880	+	3626	11.26	+	10 592	15.02	+	19 143	30.07	1.18
casw(6)	28 812 +	713 401	65.50	28 812	1.26	1 270 080	+	314 966	602.43	+	962 917	968.99	+	1 702 307	2117.97	48.52

casw(N): Thread $i \in [1..N]$ performs $\mathsf{CAS}_{\mathtt{rlx}}(x, 0, i)$; $x_{\mathtt{rel}} := i + 3$.

Results for the second set of benchmarks are shown in Table 2. On these benchmarks we see that CDSChecker considers a significant number of infeasible executions; in fact, several orders of magnitude more than the number of executions that need to be explored. Due to the exploration of infeasible executions, the times explode and some of the entries are missing; for example, indexer(15) did not finish even after running for two days. The huge number of infeasible executions seems to be related to the way CDSChecker handles $(\mathsf{sb} \cup \mathsf{rf})$-cycles, which are disallowed in RC11, and release-sequences, whose definition was corrected in RC11 following Vafeiadis et al. [2015]. Changing the ainc and binc to use release/acquire accesses yields much faster verification times, albeit still non-optimal. Nidhugg's source-DPOR algorithm performs very well in these benchmarks and the tool explores the optimal number of traces in all modes (SC, TSO and PSO) on two of the benchmarks. In contrast, a small number of sleep-set blocked (SSB) traces are explored on binc. RCMC's case is similar: its algorithm is optimal on only two of the benchmarks. On binc it explores duplicate executions. Still, their number is only a fraction of the total number, and RCMC manages to outperform Nidhugg timewise even on this benchmark.

The casw benchmark shows a different situation; cf. Table 3. Here Nidhugg encounters many SSB traces, different for each model, while RCMC is optimal (i.e., it does not examine duplicate executions) in both models. For the WRC11 model, which does not track the modification order, the traces examined are very few and coincide with the traces that CDSChecker also finds as feasible. In contrast, for RC11 the number of traces is significant and coincides with the number of non-SSB traces that Nidhugg also examines; these traces are shown on a column common for both tools.

Table 4. Results from two variants of a benchmark; Traces are the same for Nidhugg and RCMC.

	CDSChecker			Nidhugg–SC			Nidhugg–TSO			Nidhugg–PSO			Nidhugg–POWER	RCMC–RC11	RCMC–WRC11
	Traces	*Time*	*Traces*	+	*SSB*	*Time*	+	*SSB*	*Time*	+	*SSB*	*Time*	*Time*	*Time*	*Time*
lastzero0(5)	64	0.01	64	+	33	0.13	+	33	0.13	+	33	0.17	5.31	0.04	0.04
lastzero0(10)	3328	0.78	3328	+	16 867	13.21	+	16 867	14.80	+	16 867	19.28	1344.56	0.26	0.26
lastzero0(15)	147 456	50.28	147 456	+	4 651 897	4974.60	+	4 651 897	5457.25	+	4 651 897	7046.75	177 030.90	14.93	14.53
lastzero1(5)	161	0.02	64	+	0	0.13	+	0	0.13	+	0	0.14	3.68	0.05	0.05
lastzero1(10)	28 966	7.03	3328	+	0	2.29	+	0	2.74	+	0	3.42	954.91	0.32	0.31
lastzero1(15)	5 775 884	2730.19	147 456	+	0	164.92	+	0	188.29	+	0	228.52	121 642.98	15.02	14.93

`lastzero(N)`: $N + 1$ threads operate on an array of atomic variables, all initialized to 0. The first thread (say thread 0) seeks the element with the highest index and zero value. Each of the other threads reads an element of the array and updates the next one, rendering this benchmark, taken from a paper by Abdulla et al. [2017], an extremely racy one.

Table 5. Performance of the three tools on a benchmark where accesses are release/acquire.

	CDSChecker				Nidhugg–SC		Nidhugg–TSO		Nidhugg–PSO		Nidhugg–POWER		RCMC–RC11		RCMC–WRC11	
	Traces	*+Redundant+*	*Infeasible*	*Time*	*Traces*	*Time*	*Traces*	*Time*	*Traces*	*Time*	*Traces*	*Time*	*Traces*	*Time*	*Traces*	*Time*
fib_bench(3)	16 794+	272 +	513	1.23	1710	0.47	2258	0.71	2258	1.05	2258	23.74	2258	0.12	2258	0.11
fib_bench(4)	900 445+	10 403 +	41 563	78.49	19 706	5.02	34 205	9.98	34 205	17.39	34 205	732.02	34 205	1.17	34 205	1.15
fib_bench(5)	48 273 776+	447 745	+2 604 357	5072.68	219 057	69.96	525 630	192.47	525 630	276.48	525 630	20 555.25	525 630	19.42	525 630	18.64

`fib_bench(K)`: The first two Fibonacci numbers are stored in two atomic variables, and two threads are performing a concurrent calculation of the $(2K + 1)$-th Fibonacci number by loading these two variables and storing their sum into one of them (each thread in a different one). A third thread reads the values of these two variables and has an assertion that fails if any of them is greater than the $(2K + 1)$-th Fibonacci number. Taken from the paper by Abdulla et al. [2015].

The lastzero benchmark unveils interesting behaviors of the tools and the algorithms they employ. We used two variants of the benchmark, called lastzero0(N) and lastzero1(N), that differ between them only in the order that their N threads are spawned. The results we got are shown in Table 4. For Nidhugg, lastzero0 exposes the fact that source-DPOR can be exponentially worse than an optimal-DPOR algorithm [Abdulla et al. 2017]. On the other hand, if the order of events is a lucky one, the source-DPOR algorithm can perform similarly to the optimal one; such is the case for lastzero1 where no SSB traces are encountered. Nidhugg-POWER explores the same number of traces for lastzero0 and lastzero1, with lastzero1 being noticeably faster than lastzero0. However, for both variants, the performance of Nidhugg-POWER is significantly inferior to that of all other tools and/or memory models. For CDSChecker, the situation is reversed: significantly more executions are explored for lastzero1 than lastzero0. In contrast, RCMC's algorithm is robust w.r.t. this variation and the tool examines the minimum number of traces in both cases, which is of course expected since the algorithm is optimal for this program that does not contain RMWs and SC atomics.

Table 5 shows results for a benchmark (fib_bench) with release/acquire accesses. This program causes CDSChecker to explore executions which are redundant, not just infeasible. More notably, it exposes a bad scalability in the algorithm that CDSChecker employs: the number of explored traces increases with two orders of magnitude at each step, while only with one for Nidhugg and RCMC. Observe that the number of traces explored is the same for RCMC and Nidhugg running on TSO and PSO. Nidhugg-POWER also explores the same traces as RCMC but is significantly slower. Since fib_bench is the only benchmark from the first Nidhugg paper [Abdulla et al. 2015] where Nidhugg is outperformed by CBMC, we also ran CBMC [Clarke et al. 2004] on this benchmark to compare the times of RCMC against them. CBMC's numbers (in seconds) were: fib_bench(3): 0.6–0.9, fib_bench(4): 2.7–7.2, fib_bench(5): 20–25.5, depending on the memory model (SC, TSO or PSO) used, so they are roughly in the same ballpark as RCMC's. In contrast, for the version of indexer(12) program that we use here, CBMC, which is a SAT-based tool, required more than 40GB of RAM and did not manage to finish after running for several days.

We also evaluated our tool on programs consisting of code taken from "real-world" code bases, such as the Linux kernel, and turned into benchmarks. Time performance results are shown

Table 6. Performance (time in seconds) on synchronization algorithms and concurrent data structures.

	CDSChecker	Nidhugg–SC	Nidhugg–TSO	Nidhugg–PSO	RCMC–RC11	RCMC–WRC11
linuxrwlocks(2)	17.96	0.22	0.25	0.33	0.08	0.08
linuxrwlocks(3)		37.65	43.41	65.14	7.71	7.78
ms-queue(2)	0.08	0.45	0.49	0.69	0.13	0.13
ms-queue(3)	7107.10	21.21	23.13	36.12	4.37	4.49
qspinlock(2)		0.11	0.11	0.11	0.06	0.06
qspinlock(3)		15.78	17.12	26.62	3.27	3.40

linuxrwlocks(N): This CDSChecker benchmark, also used by Norris and Demsky [2016], is a reader-writer lock implementation ported from the Linux kernel. The benchmark has been adapted to be parametric on the number N of threads, which read and/or write a shared variable while holding the respective lock.

ms-queue(N): Also taken from CDSChecker, this benchmark is an implementation of the Michael and Scott queue. There are N threads, each of which enqueues and (possibly) dequeues an item into/from the queue.

qspinlock(N): A queued spinlock implementation extracted (as is) from the Linux kernel (v4.13.6). Definitions for kernel primitives, macros, and `Kconfig` options have been provided as necessary, with the testcase infrastructure occupying about 1200LoC. Queued spinlocks are the basic spinlock implementation currently used in the Linux kernel, rendering the code in this testcase heavily deployed in production. The implementation is non-trivial, since it is based on an MCS lock, but tweaked in order to further reduce cache contention and the spinlock's size (it fits in only 32 bits). In this testcase, each of the N threads writes to a shared variable while holding the lock.

Table 7. Performance of RCMC vs Nidhugg on programs where all accesses are sequentially consistent.

	Nidhugg–SC	RCMC–RC11	RCMC–WRC11
linuxrwlocks_sc(2)	0.22	0.08	0.08
linuxrwlocks_sc(3)	37.65	7.58	7.55
ms-queue_sc(2)	0.45	0.13	0.13
ms-queue_sc(3)	21.21	4.34	4.34
qspinlock_sc(2)	0.11	0.06	0.06
qspinlock_sc(3)	15.78	3.34	3.35

in Table 6. These benchmarks require some mechanism that ensures their finite execution. For CDSChecker we used `-m 2 -y`, while for Nighugg and RCMC we used `-unroll=N+1` (where `N` is the parameter of the benchmark) and **`assume`** statements. Since these mechanisms are different, the comparison of CDSChecker with the other two tools is not fair; we only include some numbers for CDSChecker in the table because two of these benchmarks originally come from its code base, and so as to add another point of reference for the rest of the numbers in the table. (Empty entries for CDSChecker are due to the tool currently not handling these benchmarks.) Although not shown in the table, in all cases the number of traces are similar for Nighugg and RCMC (and vastly different for CDSChecker).

Concentrating on the Nidhugg and RCMC columns of Table 6, we can draw three conclusions: (1) Both tools handle "real-world" code quite well. (2) RCMC is faster than Nidhugg by a factor which is consistent with that of the previous tables, whose benchmarks were synthetic but, arguably, considerably more challenging. (3) WRC11 and RC11 explore the same number of traces and have similar performance, which is in accordance with our claim that unordered concurrent writes to the same location seldom appear in real programs.

Finally, we report on the effectiveness of our approach on variants of these three benchmarks where all accesses were made SC. As shown in Table 7, in terms of verification time, RCMC outperforms Nidhugg-SC in all cases. One could argue that, since our algorithm (i) is not optimal in the presence of RMWs and SC atomics, and (ii) only checks for SC consistency when an error is detected (cf. Algorithm 1), it could be the case that RCMC becomes much slower than Nidhugg for programs with SC-only accesses. However, even if consistency checks are performed at the end of *every* exploration (this can be enabled with an RCMC switch for debugging purposes), RCMC still outperforms Nidhugg: RCMC-RC11 needs 10.09, 6.72, and 4.40 seconds for linuxrwlocks_sc(3), ms-queue_sc(3), and qspinlock_sc(3) respectively. (The times are similar for RCMC-WRC11.) As a rule of thumb, RCMC seems to slow down about 2–3 times when consistency checks are performed in every execution but, for all SC programs we have tried so far, RCMC is faster than Nidhugg-SC by a bigger factor anyway.

8 RELATED WORK

Tools like Verisoft [Godefroid 1997, 2005] and CHESS [Musuvathi et al. 2008] have paved the way for stateless model checking (SMC) and (dynamic) partial order reduction techniques to be used in software code bases of considerable size, but these tools did not consider effects of weak memory. Abdulla et al. [2015] proposed an SMC algorithm for TSO and PSO, which extended the source-DPOR algorithm for SC [Abdulla et al. 2014] by replacing the classic notion of Mazurkiewicz [1987] traces with a new partial order representation called *chronological traces*, and implemented this algorithm in Nidhugg. Subsequently, Nidhugg was extended to also handle the POWER model using a technique [Abdulla et al. 2016] that derives an operational model suitable for SMC from an existing axiomatic model, which is defined in terms of Shasha and Snir [1988] traces. In that operational model, each state is a partially constructed Shasha-Snir trace, and each step adds an instruction to the state appropriately. The algorithm is proven "optimal" in the sense that it explores each complete Sasha-Snir trace exactly once, but on the other hand it may also explore superfluous incomplete Sasha-Snir traces. The paper claims that the amount of wasted effort by such a mixture of an operational and an axiomatic approach is rare in practice [Abdulla et al. 2016], but our experimental results in Tables 1, 4 and 5 show that this is not the case — at least not for the implementation of that algorithm in Nidhugg-POWER.

A DPOR algorithm for soundly reducing the state space during runtime verification of programs for TSO and PSO has also been proposed by Zhang et al. [2015], and implemented in the rInspect tool for C/C++. The algorithm refines the dependent set to allow the reordering and introduces shadow threads to simulate the non-determinism of independent events by each thread.

Norris and Demsky [2016] developed CDSChecker, a model checker for C/C++11 that employs a variation of the classic (non-optimal) DPOR algorithm. It targets, however, a somewhat different memory model than RCMC. Its model is closer to the original C/C++11 model [Batty et al. 2011], whereas we have incorporated the improvements of Vafeiadis et al. [2015] and Lahav et al. [2017]. Nevertheless, since CDSChecker cannot generate executions with causal dependency cycles as allowed by Batty et al. [2011], it requires that deps $\cup$ $\mathtt{rf}$ is acyclic in consistent execution graphs, where deps is the subset of $\mathtt{sb}$ that has explicit syntactic dependencies. This a weaker condition than RC11's $\mathtt{sb} \cup \mathtt{rf}$ acyclicity, but one that is harder to implement, which is why the CDSChecker implementation has a few caveats, which are discussed by Norris and Demsky [2016]. The other most significant difference between the models is the definition of release sequences; our condition is straightforward to handle, whereas the C/C++11 one is rather difficult because it violates prefix-closedness of consistent executions.

SAT-directed stateless model checking approaches have also been explored and implemented in the SATCheck tool by Demsky and Lam [2015]. SATCheck is a branch-driven approach that aims to cover all branches and all the unknown behaviors of the uninterpreted functions by systematically exploring thread schedules under SC and TSO.

Bounded model checking (BMC) techniques have also been extended to handle weak memory consistency. The technique of Alglave et al. [2013a], implemented in CBMC, makes use of the fact that the trace of a program under a weak memory model can be viewed as a partially ordered set, which results in a BMC algorithm aware of the underlying memory model when constructing the SMT/SAT formula. Another work from the same group [Alglave et al. 2013b] reduces the verification problem of a program under weak memory model to verification under SC of a program constructed by a code transformation.

Kähkönen et al. [2015] and Rodríguez et al. [2015] use unfoldings [McMillan 1995] that, similarly to an optimal-DPOR algorithm and our approach, can also obtain optimal reduction in number of explored program executions in some cases. However, unfolding-based techniques have significantly larger cost per test execution than DPOR-based techniques and our approach.

Finally, Huang [2015] has proposed *maximal causality reduction* (MCR) to improve on DPOR, and recently extended it to also handle TSO and PSO through relaxed happens-before modeling [Huang and Huang 2016]. The key insight behind MCR is that a thread's behavior does not depend on the specific stores that the thread's loads take its values from, but rather on the values that these loads read. MCR uses an SMT solver to generate executions in which the loads of a thread read different combinations of values than previously explored executions. It remains to be seen whether MCR can be extended to the RC11 memory model, which, unlike TSO/PSO, does not enforce multi-copy atomicity, and how it can perform against an approach such as ours.

9 CONCLUDING REMARKS

In this paper, we have developed an effective model checking approach for RC11, which outperforms the state-of-the-art model checkers for different memory models on a variety of benchmarks. Our approach relies heavily on the fact that RC11-consistency is *prefix-closed* (see Lemma 3.8). In the absence of this property, it is not clear to us how to provide a similarly effective model checking algorithm. In addition, we rely on RC11-preconsistency, which ignores the strong guarantees provided by SC accesses, being *prefix-determined* (see Lemma 3.9). Thus, perhaps surprisingly, but in accordance with earlier observations for an alternative approach by Alglave [2013], in our approach we observe that model checking under weak memory semantics is actually easier than handling SC. This is in contrast to interleaving-based approaches, where weak memory effects are known to make the model checking problem much more difficult.

An obvious item of future work is to improve our algorithm to also handle RMWs and SC atomics optimally. Another is to try to improve the running time of the algorithm for programs, such as casw (see Table 3), with write-write races, without allowing violations of coherence. An idea could be to use the "writes-before" (wb) relation of Lahav and Vafeiadis [2015], which allows us to check that an execution is coherent without recording mo. Adopting this idea, however, is by no means trivial. In particular, (pre)consistency using wb is not prefix-determined, which means that when cutting a graph because of some revisited read, reads-from edges that previously violated coherence may now be allowed, and so they would have to be reconsidered. Moreover, wb is also not a complete solution in the presence of SC atomics: given only sb and rf, checking for SC-consistency is NP-complete [Gibbons and Korach 1997], and recording mo is what makes it tractable.

ACKNOWLEDGMENTS

We thank Brian Demsky, Marko Doko, and the POPL'18 reviewers for their helpful feedback. The second author worked on this project while at MPI-SWS supported by an ERC Consolidator Grant for the project "RustBelt" (grant agreement no. 683289). The research time of the third author was partly supported by Vetenskapsrådet (the Swedish Research Council).

REFERENCES

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

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *TACAS 2015 (LNCS)*, Vol. 9035. Springer, Berlin, Heidelberg, 353–367. https://doi.org/10.1007/978-3-662-46681-0_28

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2017. Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction. 64, 4, Article 25 (Sept. 2017), 49 pages. https://doi.org/10.1145/3073408

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

Jade Alglave. 2013. Weakness is a virtue (position paper). In *EC*^{2}. http://www0.cs.ucl.ac.uk/staff/j.alglave/papers/ec213.pdf

Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2013b. Software Verification for Weak Memory via Program Transformation. In *ESOP 2013*. Springer-Verlag, Berlin, Heidelberg, 512–532. https://doi.org/10.1007/978-3-642-37036-6_28

Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013a. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *CAV 2013 (LNCS)*, Vol. 8044. Springer, Berlin, Heidelberg, 141–157. https://doi.org/10.1007/978-3-642-39799-8_9

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *ESOP 2015 (LNCS)*, Vol. 9032. Springer, Berlin, Heidelberg, 283–307. http://dx.doi.org/10.1007/978-3-662-46669-8_12

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *POPL 2011*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *MSPC 2014*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2618128.2618134

Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On Verifying Causal Consistency. In *POPL 2017*. ACM, New York, NY, USA, 626–638. https://doi.org/10.1145/3009837.3009888

Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. 1983. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logics Specification: A Practical Approach. In *POPL 1983*. ACM Press, 117–126. https://doi.org/10.1145/567067.567080

Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *TACAS 2004 (LNCS)*, Vol. 2988. Springer, Berlin, Heidelberg, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

Brian Demsky and Patrick Lam. 2015. SATCheck: SAT-directed Stateless Model Checking for SC and TSO. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 20–36. https://doi.org/10.1145/2814270.2814297

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

Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (Aug. 1997), 1208–1244. https://doi.org/10.1137/S0097539794279614

Patrice Godefroid. 1997. Model Checking for Programming Languages using VeriSoft. In *POPL 1997*. ACM, New York, NY, USA, 174–186. https://doi.org/10.1145/263699.263717

Patrice Godefroid. 2005. Software Model Checking: The VeriSoft Approach. *Formal Methods in System Design* 26, 2 (March 2005), 77–101. https://doi.org/10.1007/s10703-005-1489-x

Jeff Huang. 2015. Stateless model checking concurrent programs with maximal causality reduction. In *PLDI 2015*. ACM, New York, NY, USA, 165–174. https://doi.org/10.1145/2737924.2737975

Shiyou Huang and Jeff Huang. 2016. Maximal Causality Reduction for TSO and PSO. In *OOPSLA 2016*. ACM, New York, NY, USA, 447–461. https://doi.org/10.1145/2983990.2984025

Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. 2015. Unfolding Based Automated Testing of Multithreaded Programs. *Autom. Softw. Eng.* 22, 4 (Dec. 2015), 475–515. https://doi.org/10.1007/s10515-014-0150-6

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP 2017*, Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

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

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *PLDI 2017*. ACM, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Antoni Mazurkiewicz. 1987. Trace Theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency (LNCS)*, Vol. 255. Springer, Berlin, Heidelberg, 279–324. https://doi.org/10.1007/3-540-17906-2_30

Kenneth L. McMillan. 1995. A Technique of a State Space Search Based on Unfolding. *Formal Methods in System Design* 6, 1 (Jan. 1995), 45–65. https://doi.org/10.1007/BF01384314

Mandanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerald Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 267–280.

Brian Norris and Brian Demsky. 2016. A Practical Approach for Model Checking C/C++11 Code. *ACM Trans. Program. Lang. Syst.* 38, 3, Article 10 (May 2016), 51 pages. https://doi.org/10.1145/2806886

Jean-Pierre Queille and Joseph Sifakis. 1982. Specification and Verification of Concurrent Systems in CESAR. In *Intl. Symp. on Progr. (LNCS)*, Vol. 137. Springer-Verlag, Berlin, Heidelberg, 337–351. https://doi.org/10.1007/3-540-11494-7_22

César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. 2015. Unfolding-based Partial Order Reduction. In *26th International Conference on Concurrency Theory, CONCUR 2015 (LIPIcs)*, Vol. 42. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 456–469. https://doi.org/10.4230/LIPIcs.CONCUR.2015.456

Dennis Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs That Share Memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (April 1988), 282–312. https://doi.org/10.1145/42190.42277

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *OOPSLA 2014*. ACM, 691–707. https://doi.org/10.1145/2660193.2660243

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. In *POPL 2015*. ACM, New York, NY, USA, 209–220. https://doi.org/10.1145/2676726.2676995

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A program logic for C11 concurrency. In *OOPSLA 2013*. ACM, New York, NY, USA, 867–884. https://doi.org/10.1145/2509136.2509532

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