Model Checking for Weakly Consistent Libraries

Michalis Kokologiannakis
MPI-SWS
michalis@mpi-sws.org

Azalea Raad
MPI-SWS
azalea@mpi-sws.org

Viktor Vafeiadis
MPI-SWS
viktor@mpi-sws.org

Abstract

We present GenMC, a model checking algorithm for concurrent programs that is parametric in the choice of memory model and can be used for verifying clients of concurrent libraries. Subject to a few basic conditions about the memory model, our algorithm is sound, complete and optimal, in that it explores each consistent execution of the program according to the model exactly once, and does not explore inconsistent executions or embark on futile exploration paths. We implement GenMC as a tool for verifying C programs. Despite the generality of the algorithm, its performance is comparable to the state-of-art specialized model checkers for specific memory models, and in certain cases exponentially faster thanks to its coarse partitioning of executions.

CCS Concepts • Theory of computation → Verification by model checking; • Software and its engineering → Software testing and debugging.

Keywords Model checking, weak memory models

ACM Reference Format:

1 Introduction

Suppose that we have a concurrent program, e.g.,

\[
\begin{align*}
  x &:= 1 \\
  y &:= 1
\end{align*}
\]

\[
\begin{align*}
  a &:= y \\
  b &:= x
\end{align*}
\]

(\text{MP})

\[
\text{assert}(a \leq b)
\]

with \(x\) and \(y\) initialized with \(0\), and we want to check whether its assertions are always satisfied. An effective way of doing so is using stateless model checking (SMC) [18, 19, 34], which enumerates all executions of the program and checks each execution individually. SMC has two major challenges.

The first challenge is associated with the memory model under which the program is executed, as it determines the program outcomes. For example, in the MP program above, the assertion \((a \leq b)\) holds under SC [28] and TSO [36], but not under PSO [40], or RC11 with relaxed’ accesses [27], because the latter two models allow for writes to distinct locations to be reordered. Thus, under PSO and RC11, \(y := 1\) may execute before \(x := 1\), thereby violating the assertion.

The second challenge is that any non-trivial concurrent program has a large number of executions that need to be explored (typically, exponential in the size of the program). To tackle this, partial order reduction techniques [1, 12, 16, 20, 42] have been developed, and try to partition the executions into equivalence classes and explore exactly one execution per equivalence class.

However, while there exist efficient techniques that target specific memory models [1–4, 12, 15, 16, 20–22, 26, 35, 39, 42], a generic technique that combats both these challenges is yet to be developed.

The goal of this paper is to develop such a model checking algorithm that is parametric in the choice of the memory model. Our algorithm, GenMC (Generic Model Checker), can be used not only for traditional memory models supporting reads, writes, and read-modify-write (RMW) instructions, but also for models incorporating high-level libraries, such as mutual exclusion locks, as primitive operations.

Our contributions can be summarized as follows:

- Through a series of examples, we present an intuitive account of our algorithm for verifying concurrent programs, using execution graphs and axiomatic semantics for any memory model (§2), so long as it satisfies four basic assumptions: porf-acyclicity, extensibility, prefix-closedness and well-blocking (§3).
- Our approach distinguishes executions based solely on the program-order and reads-from relations (§2.5), which can lead to exponentially fewer explorations compared to approaches that maintain a total coherence order between conflicting writes (§6.3).
- We demonstrate how our technique can verify programs under memory models that incorporate high-level libraries, such as mutual exclusion locks (§2.7).
- We describe our algorithm in detail (§4), and prove that it is (a) sound: produces no false positives; (b) complete: explores all possible program behaviours; and (c) optimal: explores each behaviour exactly once.
To do this, we require that the underlying memory model satisfy the following (see §3):

**MM1:** $\text{po}$ is irreflexive, where $\text{po} \subset (\text{po} \cup \text{rf})^+$

This requirement is satisfied by several models (e.g., SC, TSO, PSO, and RC11), and ensures that loop-free programs have finitely many executions. Without this requirement, we can easily run into problems as the following program illustrates:

$$x := y \parallel y := x$$  \hspace{1cm} (LB+DEP)

Under the (arguably useless) memory model that deems every execution graph consistent, the program can return $x = y = v$, for any value $v$, by having both threads read $v$ and write $v$ in a circular fashion as shown below:

\[
\begin{align*}
R(y) & \downarrow R(x) \\
W(x, v) & < \triangledown \downarrow W(y, v)
\end{align*}
\]

In the weak memory literature, such executions are considered problematic because they generate values "out of thin air" (OOTAs) \cite{10,31,41} and inhibit compositional reasoning.

**Remark 1.** While restricting OOTA behaviours, MM1 also precludes models allowing the outcome $a = b = 1$ for the following "load buffering" litmus test:

\[
\begin{align*}
a & := y; \\
b & := x; \\
x & := 1; \\
y & := 1
\end{align*}
\]

A few models allow this outcome and yet avoid OOTA executions. The Power \cite{6} and ARM \cite{37} models record (syntactic) dependencies in executions and forbid dependency cycles, while the Promising \cite{23} and WeakestMO \cite{11} models are not even defined in terms of execution graphs. Handling these models is beyond the scope of this paper.

### 2.1 Checking Consistency at Every Step

Even without OOTA executions, generating all executions and then checking consistency does not scale \cite{24}.

A much better approach, followed by most tools (e.g., \cite{1,2,4,24,35}), is to construct executions incrementally by adding events one at a time and checking for consistency at each stage, thereby avoiding the exploration of inconsistent graphs. For this approach to work, the underlying memory model must satisfy the following condition:

Every non-empty consistent graph has a po-maximal event that, if removed, yields a consistent graph.

This condition ensures that each execution can be generated by adding its events in some total extension of the po-order, and checking for consistency after each step. For instance, execution 2 in Fig. 1 can be generated by adding its events in the following order: $W(x, 0), W(x, 1), R(x)$, and $W(x, 2)$.

### 2.2 Fixing the Graph Construction Order

To generate all executions of a program following the condition of §2.1 one must in principle consider all possible
extensions of porf. This, however, very often leads to duplicate explorations.

Therefore, ideally, one would generate all executions without considering all possible extensions of porf, regardless of the memory model. In fact, we can do this for all well-known memory models. In particular, models such as SC, TSO, PSO, and RC11 all satisfy an even stronger guarantee, namely prefix-closedness:

**MM2:** There exists a partial order $R$ that includes reads-from and (preserved) program order such that, if a graph is consistent, so is every $R$-prefix of it.

This ensures that to generate a particular execution, it is sufficient to consider any total extension of porf.

As we demonstrate below, we can leverage this fact and fix an order in which we add execution events one at a time, thus generating all executions of a program systematically.

### 2.3 GenMC: A First Example

Let us run our model checking algorithm, GenMC, to generate the executions of $W^{+\text{RW}^{+}\text{W}}$ by adding its events in a fixed order given by thread identifiers: first the events of the (left-most) thread 1, then the events of thread 2, and so forth.

We start with an initial graph $G_0$ containing only the initialization write $W(x, 0)$ (see below). First, we add the $W(x, 1)$ write of the first thread to $G_0$, simultaneously adding the appropriate po edge between the events:

$$W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 1)$$

Continuing in thread order, we next add the $R(x)$ read of thread 2, which may read from either of the writes in the graph, yielding two distinct graphs (one for each case):

$$W(x, 0) \rightsquigarrow \begin{cases} W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 1) \rightsquigarrow W(x, 1) \rightsquigarrow W(x, 1) \rightsquigarrow W(x, 1) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigaro... \end{cases}$$

However, recording both graphs is inefficient: in the general case, we need to record one graph for each of the reads-from options of each read. Note that the two graphs are identical up to the read, which is the point of divergence. As such, each time we add a read that can read from more than one place, we proceed with one of the options, e.g., $W(x, 0)$, and record the alternative(s), i.e., $W(x, 1)$, into a work list $W$ for later exploration. $W$ maps each read to a list of writes it can also read from; in this case, the current graph along with $W$ is given below. We refer to revisit options such as $W(x, 1)$ as forward revisits since they are already in the graph when the read $R(x)$ is added to the graph.

$$W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 1) \rightsquigarrow W(x, 1) \rightsquigarrow W(x, 1) \rightsquigarrow W(x, 1) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigarrow W(x, 2) \rightsquigaro...$$

Since the value read is 0, we next add the $W(y, 1)$ write of thread 2. Finally, we add $W(x, 2)$ of thread 3 which yields the graph below. Note that, as it is consistent for the read to read 2 from this newly added write, we also record this new reads-from as a revisit option in $W$. We refer to revisit options such as $W(x, 2)$ as backward revisits since they are added to the graph after the corresponding read $R(x)$.

$$W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigaro...$$

This first execution is now completed (denoted by the highlighted background): it corresponds to execution (1) of Fig. 1. To generate the remaining executions, we revisit the graph by picking an alternative reads-from option from $W$.

Suppose that we next pick $W(x, 2)$ from $W$. To continue, we restrict the graph to contain only the events added to the graph prior to (and including) the read (i.e., $W(x, 0)$, $W(x, 1)$ and $R(x)$), as well as the events that led up to (in porf order) the revisiting write $W(x, 2)$. This yields the complete graph below, corresponding to execution (2) in Fig. 1. The $W(x, 2)$ option is marked as ✓ to denote that it has been considered.

$$W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigaro...$$

When considering the alternative reads-from option $W(x, 2)$, we restrict the graph to contain only the events added prior to the read. Restricting the graph is important because events added after the read may depend on its value. For instance, it is crucial to remove $W(y, 1)$ as it is only present when 0 is read from $x$. Similarly, we must retain the events added before (in porf order) the alternative reads-from option $W(x, 2)$. For instance, if $x := 2$ in $W^{+\text{RW}^{+}\text{W}}$ is wrapped in the conditional if $y = 1$ then, the presence of the $W(x, 2)$ event in the graph depends on the value read for $y$, i.e., the events before $W(x, 2)$ in porf order.

To generate the last execution, we revisit the graph once again by picking the remaining option $W(x, 1)$ in $W$. We then restrict the graph as before, yielding the graph below:

$$W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigaro...$$

To continue, we add the $W(x, 2)$ write arriving at the graph below, corresponding to execution (3) in Fig. 1. Note that we do not re-add this write as an entry in $W$, as this option has already been explored (✓-marked).

$$W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigarrow W(x, 0) \rightsquigaro...$$
Finally, as the graph is complete, and all options in $W$ are explored, the algorithm terminates.

**Avoiding Duplication** When revisiting a read event, write events may be removed from the graph and later re-added. As such, additional care is required to avoid duplicate backward revisits. For instance, continuing from (Ex2), by picking the next option in $W\ (W(x, 1))$, we removed $W(x, 2)$ arriving at (Pre3). We later re-added $W(x, 2)$ and obtained (Ex3). In doing so, we did not re-add $W(x, 2)$ as a (backward) revisit option to $W$ as this option had already been explored before. Rather, by having previously marked $W(x, 2)$ as explored (marked), we ascertained that $W(x, 2)$ is indeed a duplicate revisit. To this end, as we describe in §4, backward options are not removed from $W$; instead they are marked as explored (e.g., in (Pre3) and (Ex3)). By contrast, forward revisits do not lead to duplication. This is because when revisiting a read (e.g., $R(x)$), only events added after the read are removed from the graph. As such, since a forward option (e.g., $W(x, 1)$) is added to the graph before the read, it is not removed from the graph, and therefore not re-added, avoiding duplication. For efficiency, we thus remove forward options from $W$ once explored (e.g., $W(x, 1)$ is removed in (Pre3) and (Ex3)).

2.4 **GenMC: Extensible Memory Models**

Note that as described in §2.3, GenMC generates all executions, even though it does not add events in porf order. This is because in cases where a read is added before the write it reads from, e.g., reading from $W(x, 2)$ in 2, the $rf$ edge is recorded as an option in $W$ once the write is added.

This then leads to the question, could events added after a read affect the consistency of the execution in a way that the write is never added and hence the alternative $rf$ option is never considered? Perhaps surprisingly, the answer is yes. For example, consider the following program under a (contrived) memory model that dictates "if a read of $y$ reads 0, then there cannot be a read of $x$ that also reads 0":

$$a := x \parallel b := y \parallel x := 42 \quad (r+r+w)$$

In this case, adding the events in thread order results in a graph where both $x$ and $y$ read 0, which is then dropped as inconsistent, and thus we cannot generate the execution where the first thread reads 42. This brings us to our third requirement on memory models, extensibility:

**MM3:** Given a consistent execution $G$, a po-maximal event can always be added to $G$ to yield a consistent execution (with an appropriate $rf$ edge when applicable).

This requirement holds for all well-known memory models, and excludes "nonsensical" memory models such as that above. In particular, under that model, the consistent execution of $r+r+w$ comprising the initialization events and $R(x)$ of the first thread reading 0 cannot be extended by adding $R(y)$ for any choice of $rf$.

2.5 **GenMC: Modification Order and Writes-Before**

Recall that using GenMC, we generated all three executions of $w+rw+w$ under SC in Fig. 1. These executions, however, do not exactly correspond to the notion of executions in the formal definition of SC: as discussed above, SC executions additionally record the modification order mo, which totally orders all writes to a given memory location. We refer to such execution (which record mo) as mo-executions.

As such, the three executions in Fig. 1 correspond to the six mo-executions depicted in Fig. 2. In this program, each execution corresponds to two mo-executions representing the two ways $W(x, 1)$ and $W(x, 2)$ could be ordered by mo.

One can of course adapt GenMC to enumerate all mo-executions, as e.g., in [24]; but doing so is wasteful because while the choice of mo can affect the consistency of an execution, it is not directly observable by the program. As long as checking for consistency is reasonably efficient, enumerating only (plain) executions is better because it searches through a space that is up to exponentially smaller.2

Now, how can we check consistency of an execution besides naively enumerating all mo possibilities? The idea is to compute the "writes-before" (wb) relation, which records the set of mo-edges whose direction is forced because of the rf-edges. Let us consider the following executions under SC:

In execution L, the $W(x, 1)$ must write-before $W(x, 2)$: otherwise, the read may only read 1, due to coherence. Of course, the initialization write writes before the writes of both threads, as it is po-before them. By contrast, in execution R, the writes of the three threads are not wb-ordered, as there is no causal ordering amongst them.

2To see that, consider an extension of $w+rw+w$ with $n$ parallel writes and one reader: that program has $n+1$ executions and $(n+1)!$ mo-executions.
We next pick the alternative option for $b$ from $W$, restrict the graph as before, and obtain the (inconsistent) execution below where both RMWs read 0. Additionally, we check whether the read being revisited (i.e., $b$) may itself generate backward revisit options for existing reads in the graph. In this case, $a$ can read from $b$ and thus $b$ is added as a revisit option for $a$. This graph is then dropped as it is inconsistent (violates RMW atomicity), as denoted by the lined-background.

Finally, we pick the remaining revisit option in $W[a]$, restrict the graph as before and arrive at execution 2.

2.7 GenMC: Model Checking for Libraries

We next explain how GenMC generalizes to models incorporating high-level (abstract) libraries. To do so, let us consider a mutex library with lock and unlock instructions.

Although the mutex library does not have conventional read and write operations, its primitives behave very much like reads and writes. Intuitively, unlock can be viewed as a write, while lock can be viewed as a read that may either read from an initial value (i.e., acquiring the mutex immediately after it is initialized), or read from an unlock instruction (i.e., acquiring the mutex after it has been released by its previous holder). As with RMWs, the mutex library requires rf-functionality: no two lock events read from the same place, capturing the exclusivity of the mutex while held.

An interesting feature of the mutex library is that the calls to lock may block if the mutex is taken. Put formally, when all writes (initialization and unlocks) in an execution have already been read-from, due to rf-functionality, when adding a read (lock) event $e$ to the graph, there may not exist a write from which $e$ could read. When this is the case, the read event $e$ blocks in that its thread cannot make progress and thus $e$ has no porf successors. Note that in such libraries rf is not necessarily total on reads. However, lock events may not block arbitrarily: a lock may block only when all writes are read from; i.e., when the mutex is taken. This brings us to our final requirement on memory models, well-blocking:

**MM4:** Given a consistent execution $G$: 1) blocking reads in $G$ have no porf successors; and 2) if $G$ contains a blocking read, then all writes in $G$ are read from.

Consider the program below with its executions in Fig. 4:

$$
\begin{align*}
\text{a : lock(l);} & \quad \text{b : lock(l);} \\
\text{a' : unlock(l);} & \quad \text{b' : unlock(l);} \\
\end{align*}
$$

(lock/2)

Note that neither lock call may block as the program contains sufficient writes: two unlocks and the implicit initialization.

Running our algorithm on this example, we add the events in order ($a, a', b, b'$) and obtain execution 1. As with FAI/2, when adding $b$ to the graph, we also consider inconsistent
reads-from options and add them to the work list, arriving at the following configuration:

$$W[a]$$  

We next pick $\bot$ as a revisit option for $a$. Since $a$ is now blocking, its thread cannot proceed and its subsequent events are skipped. We thus next add $b'$ to the graph. As $b'$ is a write, it may revisit $a$ and is added as an option in $W[a]$. However, adding $b'$ renders the graph inconsistent (a blocking despite the available $b'$) and is thus dropped:

Finally, we consider the last revisit option ($b'$) for $a$. After restricting the graph, we add event $a'$ and obtain (2) in Fig. 4.

Note that running GenMC on LOCK/2 was no different from running it on FAI/2 and required no special treatment: we merely used the lock library consistency check rather than that of RC11. Indeed, the main difference between the two examples is the blocking behaviour of locks, which is prescribed by the lock library specification. As such, GenMC can be adapted to any memory model that meets the conditions in MM1-MM4. We next formalize these conditions.

### 3 Formal Model

We describe a framework for axiomatic memory models (MMs) and instantiate it to specify a mutex library. In the technical appendix [25], we present the SC [28], TSO [36] and RC11 [27] models as instances of this framework.

**Execution Graphs** The traces of a program are represented as a set of execution graphs, where each graph $G$ comprises: (i) a set of events; and (ii) a number of relations on events.

An execution is a tuple of the form $(\langle i, n, l \rangle, \emptyset)$, where $\langle i, n, l \rangle \in \text{Loc} \times \mathbb{N} \times \text{Lab}$ is the start location inside a thread, and $l \in \text{Lab}$ is an execution label. The serial number of an event denotes its index (from 1) within its thread; e.g., the first event of a thread has serial number 1. Serial number 0 is reserved for initialization events. A label may be either: (i) the error label $\text{error}$ (denoting assertion violations); or (ii) the stuck label $\text{stuck}$ (e.g., due to a failed $\text{assume}$ statement); or (iii) a memory model-specific label, e.g., the write label $\text{w(x, 1)}$ for writing 1 to $x$ under the SC model. The label function $\text{lab}$ returns the label of an event. We assume a set of locations Loc; the $\text{loc}$ function returns the location of a label.

**Definition 3.1 (Executions).** Given designated sets of read ($R$) and write ($W$) events, an execution is a tuple $G=(E, rf)$, where $E$ is a sequence of events, and $rf : E \cap R \rightarrow E \cap W$ is the reads-from function.

The sets of read and write events are designated by the memory model and are not necessarily low-level reads/writes. For instance, in case of the mutex library, lock and unlock events constitute read and write events, respectively.

Recall from §2.2 that to generate program executions using our algorithm, it suffices to fix the construction order. This is given by the order of events in the sequence $E$.

Given an execution $G$, we write $G.E$ and $G.rf$ for its components, and write $G.R$ (resp. $G.W$) for $G.E \cap R$ (resp. $G.E \cap W$). We write $G.E_i$ for $\{\{i', \sim, \sim\} \in G.E \mid i = i'\}$; and write $G.po$ for the program order defined as follows:

$$G.po \triangleq G.E_0 \times (G.E \setminus G.E_0) \cup \left\{\langle i_1, n_1, l_1 \rangle, \langle i_2, n_2, l_2 \rangle \in G.E \setminus G.E_0 \mid i_1 = i_2 \wedge n_1 < n_2 \right\}$$

In general, $G.rf$ may not be a total function: read events that do not read from any event are used to model blocking library events, such as a blocking lock event that is awaiting the release of a mutex. We write $G.B \triangleq G.R \setminus \text{dom}(G.rf)$ for the set of blocked events. Finally, although $G.rf$ is a function, we often implicitly coerce it to a relation on $W \times R$.

**Notation** Given a relation $r$ and a set $A$, we write $r^r$, $r^+$ and $r^*$ for the reflexive, transitive and reflexive-transitive closure of $r$, respectively. We write $\text{dom}(r)$ and $\text{rng}(r)$ for the domain and range of $r$, respectively. We write $r^{-1}$ for the inverse of $r$; $r_A^r$ for $r \cap (A \times A)$; and $[A]$ for the identity relation on $A$: $\{\{a, a\} \mid a \in A\}$. Given relations $r_1$ and $r_2$, we
write \( r_1; r_2 \) for \( \{ (a,b) \mid \exists c. (a,c) \in r_1 \land (c,b) \in r_2 \} \), i.e., their relational composition. Given an event set \( E \), we write \( E_x \) for \( \{ e \in E \mid \text{loc}(e) = x \} \), and \( G|_E \) for \( (E', G.\text{rf}|_E) \) with \( E' \not= G.E \cap E \). We write \( G.\text{porf} \) for \( (G.\text{po} \cup G.\text{rf})^+ \), and write \( G.\text{rf}[r \mapsto w] \) for the graph obtained from mapping \( G.\text{rf}(r) \) to \( w \). Finally, we write \( \oplus \) for sequence concatenation.

**Extension** We define graph extension in Def. 3.2, used by the incremental construction in GENMC, which describes adding an available event to an execution. Given an execution \( G \), an event \( (i,n,-) \) is available when thread \( i \) contains \( n - 1 \) events, none of which are blocking. As executions are constructed incrementally by adding one available event at a time, it follows that the events of each thread \( i \) are indexed with adjacent integers \( 1 \cdots |G.E|\).

**Definition 3.2 (Extension)** An event \( e=(i, n, l) \) is available for an execution \( G \) if \( |G.E| = n - 1 \) and \( G.E_i \cap G.B = \emptyset \). The extension of \( G \) with an available event \( e \), written \( \text{Add}(G, e) \), denotes the execution \( \langle E + [e], G.\text{rf} \rangle \).

**Consistency and Memory Model Assumptions** Given a program \( P \), the admissible behaviours of \( P \) are commonly described as a set of consistent executions. Consistency of an execution is memory model (MM)-specific; as such, MMs often define a consistency predicate that prescribes the conditions required for consistency. As our model checking technique is MM-parametric, we assume the existence of such a consistency predicate: given an execution \( G \), we write \( \text{cons}_m(G) \) to denote that \( G \) is consistent under memory model \( m \).

Recall from §2 that we require underlying memory models to satisfy certain properties as outlined by MM1-MM4. In what follows, we formally define these conditions.

The first condition (MM1) is captured by Def. 3.3. This well-formed condition additionally requires that the MM be agnostic to the order in which events are added to the graph, as it constitutes auxiliary instrumentation used by our algorithm. As such, execution consistency must be independent of this order: if \( \langle E, \text{rf} \rangle \) is consistent then \( \langle E', \text{rf} \rangle \) is also consistent, where \( E' \in \text{perm}(E) \) is a permutation of \( E \).

**Definition 3.3 (Well-formedness)** An execution \( G \) is well-formed if \( G.\text{porf} \) is irreflexive. A memory model \( m \) is well-formed iff for all \( G \), if \( \text{cons}_m(G) \) holds, then \( G \) is well-formed, and \( \forall E \in \text{perm}(G.E) \). \( \text{cons}_m(E, G.\text{rf}) \).

The prefix-closedness condition (MM2) is captured by Def. 3.4. A consistency model \( m \) is commonly considered prefix-closed iff: given a consistent execution \( G \) and a porf-closed set of events \( E \subseteq G.E \) (i.e., \( \text{dom}(G.\text{porf}; [E]) \subseteq E \)), restricting the graph to those events in \( E \) yields a consistent execution, i.e., \( \text{cons}_m(G|_E) \). However, this definition is too strong due to blocking reads.

To see this, consider the program \( l_1 : \text{lock}(l) \parallel l_2 : \text{lock}(l) \). Under the mutex specification described in §2.7, one consistent execution of this program is a graph \( G \) in which \( l_1 \) reads from mutex initialization, whilst \( l_2 \) blocks. Let \( E = \{ l_2, \text{init} \} \); if we now restrict \( G \) to \( E \), the resulting graph is inconsistent since \( l_2 \) blocks despite the available initialization event.

We thus weaken prefix-closedness by requiring that there exist a set of blocking events \( B \subseteq E \) such that the graph restricted to \( E \setminus B \) is consistent: \( \text{cons}_m(G|_{E \setminus B}) \). For instance, in the example above we can pick \( B = \{ l_2 \} \). Note that for well-known memory models such as SC, TSO and RC11, the strong and weak notions of prefix-closedness coincide, as these models do not contain blocking events.

**Definition 3.4 (Prefix-closedness)** A memory model \( m \) is prefix-closed iff for all \( G, E \subseteq G.E, \text{if dom}(\text{porf}; [E]) \subseteq E \) and \( \text{cons}_m(G) \), then there exists \( B \subseteq G.B \) such that \( \text{cons}_m(G|_{E \setminus B}) \).

Memory model extensibility (MM3) is captured in Def. 3.5 and requires that a memory model be read-, write- and rw-extensible. The first two requirements are intuitive and stipulate that a consistent execution can always be extended by a read or write event, respectively. The rw-extensibility imposes certain conditions on events that are both read and write events (e.g., RC11 RMW events). These requirements are rather technical and are necessary for the correctness of our algorithm (see the technical appendix [25]).

**Definition 3.5 (Extensibility)** A memory model \( m \) is read-extensible iff for all \( G, r, w \in \text{Add}(G, r), \text{if cons}_m(G) \), there exists \( w \in G.W \cup \{ \bot \} \) such that \( \text{cons}_m(G.\text{rf}[r \mapsto w]) \).

A memory model \( m \) is write-extensible iff for all \( G, w \in G.W \), if \( \text{cons}_m(G|_{G.E\setminus\{w\}}) \) and \( \text{rng}(\{w\}; G.\text{porf}) = \emptyset \), then \( \text{cons}_m(G) \).

A memory model \( m \) is rw-extensible iff for all \( G, r, w, u, \text{if cons}_m(G), u, u' \in G.R \cap G.W \) and \( \text{rng}(\{u\}; G.\text{po}) = \emptyset \), then:

- if \( \langle u, r \rangle \in G.\text{rf} \) and \( \text{rng}(\{r\}; G.\text{po}) = \emptyset \), then there exists \( w \in G.E \setminus \{ u \} \) such that \( \text{cons}(G.\text{rf}[r \mapsto w]) \); and
- if \( \langle w, u \rangle, \langle u', u \rangle \in G.\text{rf} \) and \( \text{rng}(\{u'; G.\text{porf} \} = \emptyset \), then \( \text{cons}(G|_{G.E\setminus\{w\}}) \).

A model is extensible iff it is read-, write- and rw-extensible.

Finally, the well-blocking condition (MM4) is captured by Def. 3.6. It stipulates that consistent executions satisfy two conditions with respect to blocking reads. First, blocking reads must be maximal in \( G.\text{porf} \): if an event blocks then it cannot proceed. Second, reads may block only when all writes are matched. That is, if there is a blocking read on \( x \) (\( G.R_x \not\subseteq \text{dom}(G.\text{rf}) \)), then all writes on \( x \) have already been read-from \( G.W_x \subseteq \text{rng}(G.\text{rf}) \). Note that when \( G.\text{rf} \) is a total function, this stipulation is trivially satisfied. As such, this is not a strong requirement: in all well-known memory models as well as the concurrent libraries specified in [38], \( G.\text{rf} \) is specified to be total.

**Definition 3.6 (Well-blocking)** A memory model \( m \) is well-blocked iff for all \( G, \text{if cons}_m(G) \), then \( G \) is well-blocked.

An execution \( G \) is well-blocked iff 1) \( [G.B]; G.\text{porf} = \emptyset \); and 2) \( \forall x \in \text{Loc.} G.R_x \not\subseteq \text{dom}(G.\text{rf}) \lor G.W_x \not\subseteq \text{rng}(G.\text{rf}) \).
From Programs to Executions Given a concurrent program, we use the same technique as [24] to pre-process it to a program of the form $P = \{ l \in \text{Tid} \mid P_i \}$, where each $P_i$ is a sequential loop-free deterministic program. The set of executions associated with $P$ is then defined by induction over the structure of sequential programs $P_i$. We omit this formal construction here as it is standard in the literature e.g., [41].

Mutex Library We formulate the notion of mutex library executions and their consistency predicate in Def. 3.7 below. For each mutex at location $l \in \text{Loc}$, the mutex events on $l$ comprise lock and unlock events, where the set of unlock events contains a single initialization event. Given a mutex execution $G = (E, rf)$, we define the mutex consistency predicate such that it holds on $G$ if: 1) $G$ is well-formed (Def. 3.3); 2) $G$ is well-blocked (Def. 3.6); 3) $E$ comprises mutex events; 4) $rf$ is injective; and 5) $rf$ maps lock events on to unlocks.

Intuitively, $rf$ describes the order of mutex acquisition. For each lock event $b$ with $(a, b) \in rf$, if $a$ is an unlock event, then $a$ denotes the event releasing the mutex immediately before it is acquired by $b$; when $a$ is the initialization event, then $b$ corresponds to the very first lock call on the mutex. As such, $rf$ must be an injection.

Note that not all locks may be matched in $rf$. Unmatched locks are blocked, waiting for the mutex release. However, well-formedness ensures that an execution contains blocking locks only when all unlock events are matched (see (2) in Def. 3.6).

Definition 3.7. The mutex event set on $l$ is $\text{MX}_l \triangleq L_l \uplus U_l$ with $L_l \triangleq \{ e \mid \text{lab}(e) = \text{lock}(l) \}$, $U_l \triangleq \{ e \mid \text{lab}(e) = \text{unlock}(l) \}$.

Execution $G$ is mutex-consistent, written $\text{cons}_{\text{mx}}(G)$, iff: 1) $G$ is well-formed; 2) $G$ is well-blocked; 3) $G.\text{E} = \bigcup_{l \in \text{Loc}} \text{MX}_l$; 4) $G.\text{rf}$ is injective; and 5) $G.\text{rf} = U_{l \in \text{Loc}} rf_l$ for some given $rf_l \subseteq U_l \times L_l$.

It is straightforward to show that $\text{cons}_{\text{mx}}(\cdot)$ is well-formed, prefix-closed, extensible and well-blocked.

4 GenMC: The Generic Model Checker

In this section, we present a version of our model checking algorithm, GenMC, that does not record $\text{mo}$. It can be instantiated for any memory model by replacing the consistency checks in the code with MM-specific consistency predicates. We refer the reader to our technical appendix [25] for a version of GenMC that also tracks $\text{mo}$.

Configurations Given a program $P$, recall from §2 that GenMC maintains a configuration comprising an execution $G$ of $P$, and a work list $W$ which stores revisit options both explored or otherwise. As described in §2.3, the options in $W$ are categorized as forward or backward revisits; forward options are removed from $W$ once explored, whilst backwards options are never removed and simply marked as explored.

Formally, we define a configuration as a tuple $(G, T, U, S)$, where $G$ is an execution of $P$; $T$ denotes a set of revisitables reads; $U$ is a map from reads to backward revisits (both explored or otherwise); and $S$ is a map from reads to both forward and backward revisits yet to be explored. As such, when a new revisit candidate is encountered, if it is a forward option, it is added only to $S$, whereas if it is a backward option then it is added to both $S$ and $U$. That is, $S$ serves as a work set (the $W$ map in §2 limited to entries not $\checkmark$-marked).

Algorithm 1 Main exploration algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>procedure Verify($P$)</td>
</tr>
<tr>
<td>2</td>
<td>$(G, T, U, S) \leftarrow \langle G_0, \emptyset, \emptyset, \emptyset \rangle$</td>
</tr>
<tr>
<td>3</td>
<td>VisitOne($P, G, T, U, S$)</td>
</tr>
<tr>
<td>4</td>
<td>while $(r, G') \leftarrow \text{RemoveMax}(S)$ do</td>
</tr>
<tr>
<td>5</td>
<td>$(E_1, r, E_2) \leftarrow \text{split}(G.\text{E}, r)$</td>
</tr>
<tr>
<td>6</td>
<td>$T \leftarrow T \setminus E_2$</td>
</tr>
<tr>
<td>7</td>
<td>$U \leftarrow U \setminus \langle U[r'] \mid r' \in E_2 \rangle$</td>
</tr>
<tr>
<td>8</td>
<td>if $G'.rf[r] \neq \perp$ then</td>
</tr>
<tr>
<td>9</td>
<td>CalcRevisits($G', T, U, S, r$)</td>
</tr>
<tr>
<td>10</td>
<td>VisitOne($P, G', T, U, S$)</td>
</tr>
</tbody>
</table>

Analogously, when there is an exploration, it is only removed from $S$ and not $U$, and thus $U$ retains all backward revisits. For efficiency, the revisitable set $T$ tracks those reads whose incoming $rf$ edges may be changed, i.e., revisit candidates.

Each entry in $S[r]$ (and $U[r]$) is a graph $G'$ representing the effect of revisiting $r$ by a write $w$. As we discuss later in §5, our implementation records only a portion of $G'$ necessary for constructing it from $G$ when $r$ is revisited by $w$. However, for better readability, in our presentation here we record in $G'$ the entire graph resulting from revisiting $r$.

The nextp Function Recall that we construct graphs by adding events in a fixed order (§2). We define a function, nextp, such that given a program $P$ and an execution $G$ of $P$, nextp($G$) returns an available event (Def. 3.2) of any thread $i$ in $G$ such that $i$ is not stuck (e.g., due to a failed assume statement) and has not finished execution. When no such thread exists (i.e., all threads are stuck or finished), nextp returns false. We implement nextp to choose the left-most such thread, i.e., one with the smallest thread identifier.

4.1 The Main Verify Procedure Given a program $P$, we begin exploring the executions of $P$ by calling Verify($P$). This routine creates an initial configuration comprising the $G_0$ graph (containing only the initialization writes), an empty revisit set $T=\emptyset$, and empty maps $U=S=S=\emptyset$ (Line 2). It then generates the executions of $P$ one at a time. This is done by calling VisitOne($P, G, T, U, S$) on Line 3, which fully explores one execution extending $G$, and pushes alternative reads-from options encountered to the work set $S$. Once VisitOne($P, G, T, U, S$) returns the full execution generated, remaining executions are generated by exploring the options in the work list $S$ Lines 4–10.
Algorithm 2 Explore one program execution

1: procedure VisitOne($P, G, T, U, S$)
2:     while cons($G$) ∧ $a ← next_p(G)$ do
3:         if $a ∈ error$ then exit("erroneous program")
4:         $G ← Add(G, a)$
5:         if $a ∈ R$ then
6:             $W ← G.E ∩ W_{loc}(a) \cup \{⊥\}$
7:         choose some $w_0 ∈ W$
8:         $G.rf[r] ← w_0$
9:         $T ← T \cup \{r\}$
10:        $S[a] ← \{G.rf[a ↦ w] | w ∈ W \setminus \{w_0\}\}$
11:    end while
12: end procedure

To do this, an option $G'$ is picked from $S[r]$ (Line 4) such that $r$ is the maximal entry in $S$: $r$ is added to the current graph $G$ after all other reads in the domain of $S$. When $S[r]$ holds multiple options, an arbitrary entry is chosen. Picking the maximal entry in $S$ makes it easier to update the current configuration and enables a key optimization (see §5).

We split $G$ at $r$ (Line 5) such that $E_1$ contains events in $G$ added before $r$ and $E_2$ contains those added after $r$. By construction, $E_2$ comprises events that either are not in $G'$ or belong to the $porf$ prefix of the event $a$ that revisited $r$ to generate $G'$. These latter events are responsible for the addition of $a$ to the graph, and consequently the reason why $r$ is revisited. As such, revisiting any of these latter events would "undo" the revisits of $r$. For this reason, we remove the events in $E_2$ from the set $T$ of revisitable reads (Line 6).

Analogously, Line 7 removes the $E_2$ entries from $U$. Note that no such entries exist in $S$: all events in $E_2$ have been added to the graph after $r$, while we picked $r$ to be the maximal entry in $S$: i.e., $S[r']$ contains no entries for $r'$ in $E_2$.

Recall from §2 that when revisiting a (non-blocked) read, we check whether the read being revisited may itself generate backward revisit options for existing reads in the graph. For instance, in the $fail/2$ and $lock/2$ examples, revisiting $b$ generated additional revisit options for $a$—see ($fail-⊥$) and ($lock-⊥$). This is done by calling CalcRevisits on Line 9. Finally, on Line 10 we explore the updated configuration.

4.2 The VisitOne Procedure

The VisitOne procedure is the workhorse of the exploration algorithm. In each iteration of this loop, while the current graph ($G$) is consistent, it is extended with its next event $a$ (given by $next_p(G)$, see page 8). When $next_p(G)$ returns $false$, VisitOne terminates. If the next event $a$ is an assertion violation, then an error is reported (Line 3), and the algorithm terminates. Otherwise, we add $a$ to the graph (Line 4). As before, we check whether the newly added event $a$ generates backward revisit options for the existing reads in the graph by calling CalcRevisits on Line 11.

Algorithm 3 Calculate which reads should be revisited

1: procedure CalcRevisits($G, T, U, S, a$)
2:     $p_a ← dom(G.porf'; [a])$
3:     for $r ∈ T ∩ R_{loc}(a) \setminus p_a$ do
4:         $⟨E_1, r, E_2⟩ ← split(G,E,r)$
5:         $G' ← G[⟨E_1, r + [r,a]⟩ ∈ E_2 ∩ p_a]$
6:         $G'.rf[r] ← \{a ∈ W \text{ then } a \perp\}$
7:     if $G' \not∈ U[r]$ then
8:         $S[r] ← S[r] ∪ \{G'\}$
9:         $U[r] ← U[r] ∪ \{G'\}$
10: end for
11: end procedure

If the new event $a$ is a write, no additional work is required. However, if $a$ is a read, we must calculate its incoming $rf$ edge. We first calculate the set of writes $W$ that $a$ could read from, i.e., its forward revisit options (Line 6). If, however, $a$ is a read, it cannot revisit existing reads in the graph itself but it may cause them to block (cf. $lock-⊥$), which is why we instead set the incoming $rf$ edge of $r$ to the blocking option $⊥$.

4.4 GenMC: Soundness, Completeness & Optimality

The GenMC algorithm (Algorithm 1) is sound, complete and optimal. Given a program $P$ and a memory model $m$, soundness ensures that if GenMC generates $G$ for $P$ under $m$, then $cons_m(G)$ holds; completeness ensures that if $G$ is an execution of $P$ under $m$ and $cons_m(G)$ holds, then GenMC generates $G$ for $P$; and optimality ensures that the $P$ executions generated by GenMC under $m$ are pair-wise distinct.

This is captured in the theorem below. The soundness proof is straightforward: GenMC checks consistency after each step, dropping inconsistent executions (Line 2 of VisitOne); as such, it only outputs consistent executions. The completeness and optimality proofs are non-trivial and are
given in full in the technical appendix [25]; we proceed with an
intuitive argument.
To show that GenMC is complete, we show that it gen-
erates all executions of a given program $P$. As discussed
in §2, MM1 and MM2 ensure that every execution of $P$
can be generated incrementally, by adding one event at a
time. We then demonstrate that each execution $G$ of $P$ generated
incrementally can also be generated by GenMC if we reshuffle
the order in which its events are added. That is, for each execution $G$
generated incrementally can also be generated by GenMC if
we reshuffle the order in which its events are added. That
is, for each execution $G$ generated by adding events in the
order $S = e_1, e_2, \ldots, e_n$, there exists a permutation $S'$ of $S$, such
that GenMC adds events in the $S'$ order and generates $G$. To
show that such a reshuffling exists, we often need to remove
events from $G$ and re-add them later (capturing the revisit
step). This can always be done thanks to the extensibility
property (MM3) ensuring that GenMC never gets stuck.
To show that GenMC is optimal, we observe that duplica-
tion can arise only when revisiting a read. As discussed
in §2 (see “Avoiding Duplication” on page 4), forward revis-
its never cause duplication since they are never removed
from the graph, while backward revisits may lead to duplica-
tion and thus the already-considered backward revisits are
recorded in the map $U$. The optimality of GenMC is thus
guaranteed by the properties of forward/backward revisits,
the map $U$ and the check on Line 7 of Algorithm 3.

**Theorem 4.1 (Correctness).** The GenMC algorithm is sound, complete and optimal.

5 Implementation

We have implemented GenMC as an open-source verification
tool for C programs over the LLVM interpreter 11i. GenMC
is available at http://github.com/mpi-swgenmc.
We have implemented three variants of GenMC:

- **LIB:** a generic variant that performs model checking on
  libraries, based on specifications provided by the user;
- **WB:** an instantiation for the full RC11 memory model [27],
  using a consistency check based on wb; and
- **MO:** an alternate RC11 instantiation that records the no order
during exploration. That is, whenever a write is added to a graph, we consider all its possible placements in no, and create subexplorations for each case.

Naturally, the generic variant is slower than the RC11 ones because the latter have more optimized consistency checks; it is, however, still optimal.

Further, we have implemented some optimizations over
the algorithm described in §4, which we will describe below.
The first key optimization has to do with the representa-
tion of the graphs to be revisited in $S$. In §4, each entry in
$S[r]$ (and $U[r]$) is a full graph $G'$ generated by a forward or
backward revisit of the read $r$. For better space efficiency,
rather than recording the entire graph $G'$, we store only the
portion of $G'$ of events after $r$, because that suffices for re-
constructing the entire $G'$ when the revisit takes place. The
reason is that GenMC revisits executions from $S$ by always
choosing the maximal read in $S$ when removing an entry
from $S$ (Line 4 of Algorithm 1). The effect of the revisit order
is that the current graph projected to the events before $r$ (i.e.,
$E_1$ in Algorithm 1) is exactly the same as the recorded graph
$G' \in S[r]$ projected to the same events. As a result, it suffices
to record in each graph in $S$ only the revisited read and the
events after it.
Simlarly, we store the graphs in $U$ in a compressed form.
Since we do not ever need to restore the graphs from $U$, we
do not need to store all the events after $r$; it suffices to record
only their incoming rf edges because those determine the
values read and hence the event labels.

Finally, in the WB and MO variants of GenMC, we use
optimized consistency checks when adding a new event to
the graph. We exploit the fact that the graph prior to adding
the event was consistent, so it suffices to check only that the
new event does not lead to any consistency violation.

6 Evaluation

**Verification Tools** In the following, we compare the per-
formance of GenMC to three other stateless model check-
ers: Nidhugg [2], RCMC [24], and Tracer [4]. Initially, we
also considered other tools—namely, CBMC [5, 14], CDS-
Checker [35], and Herd [6]—but exclude them from head-
to-head comparisons because they are typically significantly slower than Nidhugg and RCMC and do not scale well (see,
e.g., the evaluation in [24]): Herd because it was meant
for experimenting only with small “litmus test” programs,
CBMC because of the SAT solver, CDSChecker because of
its suboptimal partial order reduction technique.
Nidhugg [2] is a state-of-the-art stateless model checker
supporting SC, TSO, and PSO. It enumerates mo-executions
(a.k.a. Mazurkiewicz traces [32]) and can operate both under
an optimal mode (optimal-DPOR) and a non-optimal mode
(source-DPOR). In our benchmarks, we use the source-DPOR
version because it is typically faster than the optimal version.
Under SC, Nidhugg can also operate under a coarser equivalence partitioning (denoted SCa – Nidhugg with ob-
servers) [7]. This equivalence can be exponentially coarser
than mo-executions, but remains exponentially finer than
plain executions. We used version 0.3 of Nidhugg, and ran
it with the --c11 switch, which makes the SC version of
Nidhugg noticeably faster.

RCMC [24] targets RC11 and WRC11, a weaker RC11 vari-
ant that does not record mo and does not enforce coherence.
RCMC-RC11 also enumerates mo-executions of a program,
though not optimally in the presence of RMW or SC accesses.

Tracer [4] targets RA (the release-acquire fragment of
RC11), and enumerates plain executions. It is, however, built
over the CDSChecker infrastructure, which makes it quite

---

6Nidhugg also provides some very limited support for POWER, which we
do not evaluate because it cannot encode most of our benchmarks.
difficult to apply it fairly to our benchmarks (e.g., it does not support assume statements, and requires manual instrumentation for programs with loops). For this reason, we apply it only to our synthetic benchmarks.

**Benchmarks** We took as benchmarks all the programs from the benchmark suites of Nidhugg and RCMC, together with some additional larger programs (e.g., seqlock, chaselev) from open-source code. In total, we have assembled 127 benchmark programs, some of which are parametric in the number of operations/threads. For suitable values for their parameters, we have generated 202 test cases in total.

First (§6.1), we focus on the generic GenMC variant, and demonstrate how it is used to model check libraries. We conduct a case study for a lock library, and show that abstracting over its implementation has substantial runtime benefits.

Next (§6.2), we evaluate the overall performance of the RC11 variant of GenMC in both synthetic and real-world benchmarks. Our benchmarks highlight the importance of our optimality result, and show that GenMC verifies code currently deployed in production within seconds.

Finally (§6.3), we perform an extensive comparison between the WB and MO variants of GenMC. We show that the WB variant can explore exponentially fewer executions than MO, and the overhead due to its more expensive consistency checks is usually negligible.

**Experimental Setup** We conducted all experiments on a Dell PowerEdge M620 blade system, running a custom Debian-based distribution, with two Intel Xeon E5-2667 v2 CPU (8 cores @ 3.3 GHz), and 256GB of RAM. We used LLVM 3.8.1 for RCMC and Nidhugg. Unless explicitly noted otherwise, all reported times are in seconds.

### 6.1 Model Checking a Lock Library

As a simple demonstration of the benefits of parametricity and compositional verification, we consider a C implementation of Lamport’s fast mutual-exclusion algorithm [29] (see Table 1). We could have considered any correct lock implementation (e.g., the ones used in §6.2), but we chose Lamport’s algorithm because it has write-write races, which are rare in non-synthetic programs and highlight the differences between the various tools. Nidhugg under TSO and PSO are excluded from this table for brevity, as they are slower than Nidhugg-SC.

<table>
<thead>
<tr>
<th>Nidhugg</th>
<th>RCMC</th>
<th>GenMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC</td>
<td>SCc</td>
<td>MO</td>
</tr>
<tr>
<td>RC11</td>
<td>WRC11</td>
<td>WB</td>
</tr>
<tr>
<td>lamport(2)</td>
<td>0.13</td>
<td>0.10</td>
</tr>
<tr>
<td>lamport(3)</td>
<td>7.53</td>
<td>4.49</td>
</tr>
<tr>
<td>lamport(4)</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

The first observation is that RCMC does not terminate under WRC11. This is because this test case has writes that are never ordered under WRC11, which makes the threads’ reads “oscillate” between the values of these writes ad infinitum. This behaviour is ruled out by RC11 and stronger memory models. Additionally, both RCMC and GenMC outperform Nidhugg-SC (even though they explore more executions), with RCMC-RC11 being faster than GenMC-MO (see §6.2).

However, by feeding the axiomatic definition of the lock library to GenMC, and abstracting the inner working of the locks, GenMC is much faster than the other tools (shown in column LIB). For N = 4, for example, all other tools take more than 3 days to complete, whereas the generic variant of GenMC terminates almost instantly.

### 6.2 Overall Performance

Table 2 reports two synthetic benchmarks, which demonstrate the importance of optimality (Table 2).

In the cinc program, all threads perform a series of RMW operations. Since RCMC is not optimal in the presence of RMWs, it can explore many more executions than necessary, which leads to some runtime overhead. For 4 threads RCMC explores 45% more executions than GenMC, while for 5 threads, it explores almost twice as many executions as GenMC, and this is reflected in the running time. All other tools explore the same number of executions, but Nidhugg is significantly slower than the other tools.

The Nw1r program has N + 1 concurrent writers and one concurrent reader of a shared variable, and thus has (N + 2)! mo-executions versus only (N + 2) plain executions. It is therefore not surprising that tools enumerating mo-executions (Nidhugg-SC, RCMC-RC11, and GenMC-MO) do not scale well. Nidhugg-SCc explores 193 executions for N=5 and 2305 for N=8, and so also does not scale particularly well. In contrast, Tracer, RCMC-WRC11, and GenMC-WB finish almost instantly. Recall, however, that RCMC-WRC11 fails to terminate on other benchmarks (§6.1).

Next, we move to two sets of benchmarks extracted from real programs. Since Nidhugg-SCc does not reduce the number of executions and is in fact slower than Nidhugg-SC on these benchmarks, we exclude it from further comparisons.

Table 3 compares the tools on the implementations of concurrent data structures from [13, 35]. We do not show the number of executions explored because all tools explore
the same number of distinct executions\(^5\), excluding possible redundant executions explored by Nidhugg (under 5%). These benchmarks have the same number of distinct executions regardless of the memory model (i.e., they are robust), which is expected since they only use non-SC accesses for performance reasons. The only exception is chase-lev, for which Nidhugg explores more executions under PSO due to the absence of a store-store fence, which renders the precise modeling of acquire-release operations utilized by this benchmark difficult.

On these benchmarks, RCMC and GenMC outperform Nidhugg, even though they operate under a weaker memory model. By contrast, Nidhugg gets slower as the memory model gets weaker, which is expected due to the way it models TSO and PSO, and agrees with the observations in\(^2,\ 24\). GenMC performs similarly in terms of time under WB and MO, and explores the same number of executions. For linuxrwlocks, however, the WB verification requires much more time than MO. This is due to the calculation of wb as part of RC11’s consistency check, which is particularly slow when there are long chains of RMW events. (In general, calculating wb can take up to \(O(n^3)\) time in the size of the execution graph, and achieves its worst-case complexity, when there are many writes to the same location.)

Table 4 summarizes the performance of the tools in lock implementations extracted verbatim from the Linux kernel (v4.13.6, v4.19.1). Headers, kernel primitives definitions, macros, and Kconfig options have been provided for all benchmarks as necessary. The test cases involve \(N\) threads accessing shared variables while holding the respective locks. For all benchmarks, except mcs_spinlock, all tools explore the same number of executions, modulo a few redundant explorations for Nidhugg, and the seqlock test case, where Nidhugg-PSO again explores more executions due to the absence of a store-store fence. As shown, RCMC and GenMC outperform Nidhugg by a large factor.

The mcs_spinlock benchmark is rather interesting for several reasons. First, it allows some relaxed behaviours to take place, and so Nidhugg-PSO, GenMC-MO, and RCMC-MO explore more executions than Nidhugg-SC and Nidhugg-TSO (approximately 15% more). Nonetheless, GenMC and RCMC outperform Nidhugg by a large factor. Second, GenMC-WB and RCMC-WRC11 explore fewer executions than GenMC-MO and RCMC-RC11, and shows the benefit of not recording mo in terms of verification time. Last, GenMC is slower than RCMC on this particular benchmark. This is because GenMC’s revisit procedure removes more events from the graph during backward revisits than RCMC. The extra events must then be re-added resulting in runtime overhead. Of course, this also depends on the nature of the benchmark, and the backward revisits that take place.

6.3 Modification Order vs Writes-Before

We next compare GenMC-WB and GenMC-MO more thoroughly. Admittedly, calculating wb for consistency is much more expensive (\(O(n^3)\)) than using the total order readily given by mo. As we show, however, (a) it can lead to exploring exponentially fewer executions than recording mo; and (b) the overhead imposed by the wb calculation is usually negligible.

To see (a), consider Fig. 5 (left), depicting the number of executions explored by GenMC-WB and GenMC-MO on some synthetic benchmarks. As shown, for 7 threads, GenMC-MO can visit up to 10\(^6\) more executions than GenMC-WB, which is also reflected in the running time.

---

\(^5\)Nidhugg counts the number of executions that contain a failed assume() statement, while RCMC does not; we take this discrepancy into account.
We have presented we can relax these assumptions to enable verification under well-blocking. In the future, we plan to investigate whether GenMC and wb can be beneficial to spot such errors.

7 Conclusions and Related Work

We have presented GenMC as an effective model checking approach that is parametric in the choice of memory model and supports high-level concurrent libraries. Our approach relies on four basic assumptions about the underlying memory model: porf-acyclicity, extensibility, prefix-closedness, and well-blocking. In the future, we plan to investigate whether we can relax these assumptions to enable verification under hardware memory models such as Power [6] and ARM [37] (that do not satisfy porf-acyclicity) and library specifications such as queues [38] (that are not prefix-closed).

Amongst the verification tools handling weak memory models (MMs), the only properly MM-parametric tool is Herd [6], a memory model simulator that allows users to experiment with different consistency predicates on small “litmus test programs. Unlike GenMC, Herd does not require models to satisfy conditions MM1-MM4, and so accepts a wider range of models than GenMC. Nevertheless, it follows the simple approach of enumerating all possible executions and filtering them according to the user-supplied consistency predicate, and thus is not scalable when applied to larger programs. It would be worth extending Herd to use the GenMC approach whenever the user-supplied model can be shown to satisfy conditions MM1-MM4.

As discussed in §2, several tools based on stateless model checking [18, 19, 34] combined with (dynamic) partial order reduction (DPOR) techniques [1, 16] have targeted specific memory models [2–4, 15, 24, 35, 42]. Unfortunately, all of them use somewhat different ideas, making it difficult to get a model checking algorithm that is MM-parametric. Amongst these tools, the only ones enumerating plain executions (as opposed to mo-executions) are: Tracer [4] for the release-acquire fragment of (R)C11; DC-DPOR [12] for SC; and RCMC [24] for the WRC11 model.

GenMC follows the general design of RCMC, but uses a revisit procedure akin to that of Tracer, i.e., when in an execution graph G a write w revisits a read r, it removes from G all events that were added to G after r and are not porf-before w as opposed to removing only the events porf-after r. As a result, the completeness proof of GenMC (unlike that of RCMC) does not require “prefix-determinacy” [24, Lemma 3.9], which does not hold for the entire RC11 model: the weaker “prefix-closedness” suffices. So, while RCMC is optimal only in the absence of RMW and SC accesses, GenMC achieves optimality for the full RC11 model.

Other tools, such as CBMC [14], encode all executions of a program together with the memory model in a SAT/SMT formula and query a dedicated solver for its satisfiability [5]. This approach should in principle be able to handle models such as RC11; however, it is currently limited to SC, TSO, and PSO. The main drawback of this approach is its SAT/SMT component, which can be slow and highly unpredictable. As a result, CBMC tends to be significantly slower than Nidhugg on relevant benchmarks [26, 30].

Another approach is maximal causality reduction (MCR) [20, 21], which introduces an even coarser equivalence partitioning than porf, based on values and not the places reads read-from. This approach fundamentally assumes “multi-copy atomicity” (i.e., that writes propagate simultaneously to all other processors), and thus cannot work for RC11 [27]. It does, however, work well for SC, TSO, and PSO.

Finally, unfolding-based techniques [22, 39] have obtained similar optimality results with some DPOR algorithms for SC. It remains to be seen whether they can be generalized or achieve optimality under a coarser equivalence partitioning.

Acknowledgments

We would like to thank Michael Emmi, Konstantinos Sagonas and the PLDI reviewers for their feedback. The second author was supported in part by a European Research Council (ERC) Consolidator Grant for the project “RustBelt”, under the European Union Horizon 2020 Framework Programme (grant agreement number 683289).


