Enhancing GenMC's Usability and Performance

Michalis Kokologiannakis, Rupak Majumdar, and Viktor Vafeiadis

MPI-SWS, Kaiserslautern, Germany
{michalis,rupak,viktor}@mpi-sws.org

Abstract. GenMC is a state-of-the-art stateless model checker that can verify safety properties of concurrent C/C++ programs under a wide range of memory consistency models, such as SC, TSO, RC11, and IMM. In this paper, we improve the performance and usability of GenMC: we provide a probabilistic estimate of the expected verification cost, we automate the porting of new memory models, and employ caching and other data structure optimazations to improve the tool's performance.

1 Introduction

GenMC [32, 27] is a state-of-the-art stateless model checker that verifies assertion safety of concurrent C/C++ programs in a fully automated ("push-button") fashion. In its core, it implements the TruSt dynamic partial order reduction (DPOR) algorithm [27], which has polynomial space complexity and optimal time complexity: it explores only the consistent executions of the given program and never repeats any work. GenMC also incorporates custom techniques for verifying programs with constructs such as synchronization barriers [31] and loops [30, 28] more effectively.

Despite its solid theoretical foundations, certain parts of GenMC's *implementation* were somewhat neglected, and are addressed as part of this work.

Time Unpredictability (§3): Non-expert users of GenMC were finding it difficult to estimate how long verification will take, and whether it is worth waiting for the verification result or give up. To address this problem, we implement a procedure that produces a probabilistic estimate of the size of the state space so that users can anticipate the total verification cost, and perhaps revise their code as necessary.

Customization Difficulty (§4): Although the TruSt algorithm is parametric in the choice of the underlying *memory consistency model* (MCM) [9], adding support for new MCMs to GenMC was arduous and required human effort. To make the tool more easily customizable, we extend an already existing domain specific language so that users can port new MCMs into the tool completely automatically.

Overall Performance (§5): GenMC used to spend a lot of time repeatedly simulating the execution of LLVM bytecode with an interpreter, which led to non-trivial performance overhead. We improve the tool's performance by caching interpreter results and optimizing other internal data structures.

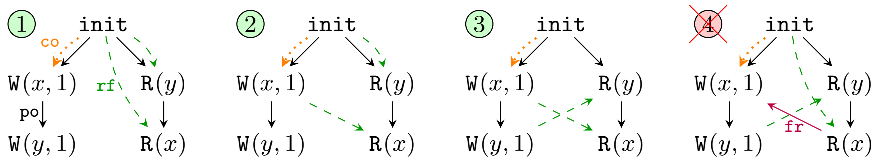

Fig. 1. MP: Three consistent and one inconsistent execution graphs under SC and RA. We omit co edges for variable y to avoid cluttering the presentation.

2 Background

We begin with a brief tour of declarative MCM semantics (§ 2.1) and proceed with a description of GenMC's core model checking algorithm (§ 2.2).

2.1 Semantics of Memory Consistency Models

When dealing with multiple MCMs, it is convenient to use *declarative semantics* to represent program executions as execution graphs. An execution graph comprises a set of nodes corresponding to program instructions (e.g., loads or stores to shared memory), and a set of edges corresponding to various relations among the instructions. Examples of primitive relations used by most MCMs are: the *program order* (po), which orders the instructions of each thread, the *reads-from* (rf) relation, which maps each read to the write it gets its value from, and the *coherence order* (co), which totally orders writes to the same memory location.

The semantics of a program P under a model M is expressed as a set of "consistent" execution graphs, representing an abstract set of program executions that the model allows. Consistency for a given MCM entails satisfying the MCM's consistency predicate.

Consistency predicates are typically expressed in relational algebra. For instance, *sequential consistency* (SC) [36], a standard MCM where threads take turns executing their instructions, demands that $(\mathtt{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr})$ be acyclic, where $\mathtt{fr} \triangleq \mathtt{rf}^{-1}; \mathtt{co}$. *Release-acquire* (RA) [34] is a weaker MCM, which demands that $(\mathtt{po} \cup \mathtt{rf})^{+}; (\mathtt{co} \cup \mathtt{fr})^{?}$ be irreflexive.

To illustrate these concepts, consider the MP example below along with the annotated outcome, which corresponds to graph 4 in Fig. 1 and is forbidden under both SC and RA.[1]

$$\begin{array}{l||l} x := 1 & a := y \mathbin{/\!/} 1 \\ y := 1 & b := x \mathbin{/\!/} 0 \end{array} \qquad \text{(MP)}$$

Graph 4 is inconsistent according to both SC and RA because of the $\mathtt{po}; \mathtt{rf}; \mathtt{po}; \mathtt{fr}$ cyclic path from $\mathtt{W}(x, 1)$. Intuitively, thread II cannot read a stale value for x after having read thread I's write to y. The other three depicted execution graphs 1, 2, 3 are consistent and correspond to the outcomes where $a = 0 \vee b = 1$.

[1] In all our examples, x, y, z are shared variables, while $a, b, c, ...$ are local.

Algorithm 1 An optimal graph-based DPOR algorithm

```
1: procedure VERIFY($P$)
2:     VISIT$_P$($G_\emptyset$)
3: procedure VISIT$_P$($G$)
4:     $a \leftarrow$ ADDNEXTEVENT$_P$($G$)
5:     if $a = \bot$ then return "Execution complete"
6:     if ISERRONEOUS$_\mathsf{M}$($G$) then exit("error")
7:     if $a \in \mathtt{R}$ then
8:         VISITRFS$_P$($G, a$)
9:     else if $a \in \mathtt{W}$ then
10:        VISITCOS$_P$($G, a$)
11:        for $r \in G.\mathtt{R}_{\mathtt{loc}(a)} \setminus \mathsf{cprefix}(a)$ do
12:            if $\neg$DUPLICATEREVISIT($G, \langle r, a\rangle$) then
13:                $Deleted \leftarrow \{e \in G.\mathtt{E} \mid r <_G e <_G a\} \setminus \mathsf{cprefix}(a)$
14:                VISITCOS$_P$($\mathsf{SetRF}(G \setminus Deleted, r, a), a$)
15:    elseVISIT$_P$($G$)
16: procedure VISITRFS$_P$($G, a$)
17:    for $w \in G.\mathtt{W}_{\mathtt{loc}(a)}$ do
18:        $G' \leftarrow \mathsf{SetRF}(G, a, w)$
19:        if $\mathsf{consistent}_\mathsf{M}(G')$ then VISIT$_P$($G'$)
20: procedure VISITCOS$_P$($G, a$)
21:    for $w_p \in G.\mathtt{W}_{\mathtt{loc}(a)}$ do
22:        $G' \leftarrow \mathsf{SetCO}(G, w_p, a)$
23:        if $\mathsf{consistent}_\mathsf{M}(G')$ then VISIT$_P$($G'$)
```

2.2 Dynamic Partial Order Reduction and GenMC

Declarative semantics enable effective automated verification with the "TruSt" model checking algorithm [27]. TruSt is a graph-based *dynamic partial order reduction* (DPOR) algorithm that takes as parameters a program P and a memory consistency model M. It verifies the program by generating all M-consistent graphs of P and checking that they do not contain any errors. For this purpose, we assume the MCM M defines three components:

1. a *causal order*, $\mathtt{corder} \subseteq (\mathtt{po} \cup \mathtt{rf})^+$, prescribing causal dependencies among the instructions;
2. the $\mathsf{consistent}_\mathsf{M}(G)$ predicate prescribing when a graph G is consistent; and
3. the ISERRONEOUS$_\mathsf{M}$(G) predicate prescribing whether the graph contains an error (e.g., an assertion violation or a data race).

The core structure of GENMC's DPOR algorithm can be seen in Algorithm 1. In what follows, we provide a high-level overview of the algorithm and refer readers to Kokologiannakis et al. [27] for a more detailed presentation.

VERIFY generates all possible execution graphs of P by calling VISIT on the initial graph $G_\emptyset$ containing only the initialization event `init`.

In turn, VISIT generates all program executions incrementally by extending a given graph with one event at a time by calling ADDNEXTEVENT(Line 4), which selects some unfinished program thread and runs an interpreter to run the code of that thread until its next event.

VISIT returns successfully if no further events can be added (Line 5, i.e., if all threads are finished or stuck) or when an error is encountered (Line 6). Otherwise, the next action that VISIT takes depends on the type of a.

If a is a read event (Line 7), VISIT recursively explores all its possible `rf` options by calling VISITRFS. The latter iterates over all same-location writes (Line 17) and recursively explores the ones that preserve consistency (Line 19).

If a is a write event (Line 9), similarly to the read case, VISIT recursively explores all possible `co` options for it by calling VISITCOS (Line 10). In addition, VISIT *revisits* existing same-location reads in G, since they did not have the chance to read from a when they were added. Specifically, for each read r that does not causally precede a (Line 11), VISIT checks (Line 12) whether a should revisit r (i.e., that the revisit has not taken place in some other exploration), and if so calls VISITCOS on an appropriately restricted graph G' (Line 14).

Restricting the graph is necessary because the value read by r might affect e.g., the control flow of the corresponding thread. In GENMC, the restricted graph only contains the events that were added before the revisited read r, as well as the ones causally preceding a, effectively creating a graph that models a scenario where a and its prefix were present when r was added. (The way a graph is restricted is important when estimating the program state space in §3.)

Finally, if a has any other event type (Line 15), VISIT recursively calls itself.

To conclude our brief presentation of GENMC's algorithm, we reiterate its core properties. Algorithm 1 is optimal (i.e., it explores each consistent execution graph of P exactly once and never embarks into futile explorations), has polynomial memory consumption, and can accommodate arbitrary MCMs subject to the following constraints:

Well-formedness: Consistency does not depend on the order in which events are added to the graph, and, in consistent graphs, `corder` should be acyclic (i.e., an event cannot circularly depend on itself).

Prefix-closedness: Restricting a consistent graph to any `corder`-prefix-closed subset of its events yields a consistent graph.

Extensibility: Adding a `corder`-maximal event to a consistent graph preserves consistency for some choice of `rf`/`co`. Intuitively, for the case that $\texttt{corder} \triangleq (\texttt{po} \cup \texttt{rf})^+$, executing a program should never get stuck if a thread has more statements to execute: each read can read from the most recent, same-location write, and each write can be added last in `co`.

3 Estimating the Program State Space

The execution time of Algorithm 1 is difficult to predict because it depends on the number of consistent execution graphs of the program P, which in turn is challenging to estimate without actually generating the consistent graphs. In

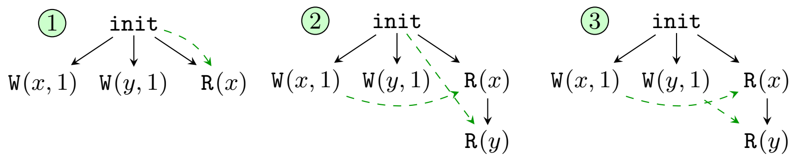

Fig. 2. R+W+W: three consistent execution graphs under SC

particular, the number of possible `rf`s/`co`s for a given read/write is not fixed beforehand, and depends on the exploration choices made so far. Moreover, since the state space is often asymmetric, we cannot estimate the remaining exploration options merely by assuming that untaken exploration options (i.e., unexplored `rf`s or `co`s) will yield the same number of exploration options as their taken counterparts.

3.1 The Basic Approach

One solution for the problem above is to use a Monte Carlo simulation [23]. In Monte Carlo methods, one generates a number of random samples from a given input domain, performs some computation on each sample, and then aggregates the results. As long as the sampling process is unbiased, the law of large numbers guarantees that the empirical mean of the obtained samples will approximate the expected value of the corresponding random variable.

Applying this idea to our problem, we construct a randomized version of Algorithm 1 that generates a single consistent program execution by adapting VisitRFsand VisitCOsto explore a random `rf`/`co` choice for each read/write (picked uniformly at random) instead of all consistent choices. For one execution, we can estimate the size of the search tree by taking the product of the exploration options encountered for each read/write (effectively assuming a symmetric state space). We can then run the modified Visit on $G_\emptyset$ a fixed number of times, and take the mean of the individual estimates as our state-space estimate.

Let us examine how such a method would work with the example below, where there are three graphs to be generated (shown in Fig. 2).

$$x := 1 \;\Big\|\; y := 1 \;\Big\|\; \begin{array}{l}\texttt{if}\ (x = 1)\\ \quad b := y\end{array} \qquad \text{(W+W+RR)}$$

Assuming the algorithm adds events from left to right, the state-space size will be estimated to be either $2 \cdot 2 = 4$ (in samples where the algorithm makes the read of x read 1 and the read of y also appears), or 2 (in samples where the algorithms makes the read of x read 0, and the read of y does not appear). In fact, since the algorithm picks among the `rf` choices for x uniformly at random, each of these estimations occurs with a similar frequency, thereby yielding an estimated mean that approximates 3 (given a sufficiently large sample).

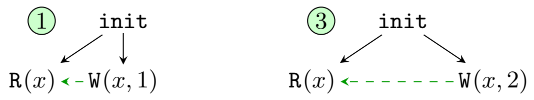

Fig. 3. R+W+W: two possible revisits

3.2 Problems with Revisits

The challenge of applying Monte Carlo method to our setting is achieving an unbiased sampling in the presence of revisits. (Recall from §2 that if a write w is added after a same-location read r, GenMC will also examine the scenario where w revisits r, but only if a certain revisiting condition C holds.)

Blindly applying this revisiting condition when estimating, however, does not work. Since its purpose is to avoid duplication, the probability that C holds for a sequence of random rfs/cos choices is tiny, thereby yielding a biased simulation.

Similarly, one cannot ignore the revisiting condition C and perform revisits with some probability p because this can lead to very long runs. A given revisit R may delete part of the execution graph, which when subsequently re-added might invoke another revisit R', which may in turn delete the write that performed R, and this process can be repeated again and again. To see this, consider the R+W+W program below and the graphs that occur as a result of each of the writes revisiting the read of x, depicted in Fig. 3.

$$a := x \;\|\; x := 1 \;\|\; x := 2 \qquad (\text{R+W+W})$$

As can be seen, if $\mathtt{W}(x, 1)$ revisits $\mathtt{R}(x)$ then $\mathtt{W}(x, 2)$ is deleted, and vice versa.

An obvious solution to this issue would be to preclude multiple revisits and e.g., only allow one revisit per sample. Even in this case, however, it is quite difficult to find the probability p with which revisits should take place. Indeed, suppose that once a revisit is performed, the revisiting write and its causal prefix can never be deleted from the execution graph. Alas, in such an approach, the probability that some read r gets revisited by a given write w decreases exponentially as the number of writes that are added after r (but before w) increases, and we again obtain a biased simulation. In the R+W+W above, $\mathtt{R}(x)$ will be revisited by $\mathtt{W}(x, 1)$ with probability p and by $\mathtt{W}(x, 2)$ with probability $(1 - p)p$. (Observe that, even if p is not fixed, the probability that later writes get to revisit a given read r decreases exponentially.)

Finally, precluding revisits altogether is not a viable solution either. Even though such an approach would yield an unbiased simulation, the simulation would only cover a subset of the input domain, as revisits would not be accounted for. In the R+W+W example, precluding revisits means that the state-space size would always be erroneously estimated to be 1.

3.3 Our Solution

Our solution to make the sampling process as unbiased as possible is twofold. First, we keep a *choice map* for each encountered event representing its possible exploration alternatives (e.g., rf or co options). The reason a map is used is to be able to account for revisiting: as possible rf options for a read r might appear after r has been added, we cannot calculate the product of all available explorations options on-the-fly. Instead, we populate the available rf options for each read dynamically, as more rf options become available, and calculate the product at the end of each complete execution.

Second, as far as revisiting is concerned, in order to maintain an unbiased estimation, we do preclude revisits, but mitigate the negative effects of this decision by employing a *custom scheduler* that prioritizes the addition of writes over that of reads (so that reads have as many rfs as possible available when they are added), and chooses uniformly at random when only reads can be added.

Intuitively, the reason this approach works well is that each of the graphs in a program's declarative semantics can be generated incrementally, following `corder` (see §2). What this means is that we can, in principle, generate all graphs in a program's declarative semantics without any backward revisiting. Of course, it can still be the case that a read is added before some of its possible rfs (e.g., if the next available event of all threads is a read), but picking at random among reads guarantees that all rf options will eventually be considered (though, perhaps, some of them not often enough).

A modification of VERIFY that implements the above approach can be seen in Algorithm 2. The ESTIMATE function obtains a number of samples[2] by calling GETSAMPLE (Line 4), and calculates the mean of all estimations (Line 6).

GETSAMPLE closely resembles VISIT, with the addition of a choice map C that stores consistent exploration options for each event of an execution graph. At each step, GETSAMPLE extends the current graph with an event a obtained by our custom scheduler (Line 8), and then performs a case analysis on a's type.

If a is a read, PICKRF populates $C[a]$ with all consistent rfs for a (Line 18), and then picks an rf for a uniformly at random (Line 19).

If a is a write, PICKCO performs actions similar to the ones taken by PICKRF in the read case (Line 14), and upon returning, GETSAMPLE also updates the entries of all revisitable reads, recording a as a possible rf. (Observe that no revisiting is performed during the estimation process.)

Finally, at the end of each sample (Line 9), GETSAMPLE returns the current estimation $\prod_{e \in G} |C[e]|$.

3.4 Stopping the Sampling Process

Before concluding the presentation of our estimation procedure, let us discuss when that estimation procedure should stop, i.e., when an adequate number of samples has been taken.

While SHOULDKEEPSAMPLING in Algorithm 2 could in principle return **false** after a fixed number of samples, doing so is often undesirable. For programs with a very small state space, it does not make sense to obtain more samples than the number of consistent executions because that would unnecessarily delay

[2] See § 3.4 for how this number is calculated.

Algorithm 2 Estimating the DPOR state space

```
1:  procedure ESTIMATE($P$)
2:      $T \leftarrow 0, Samples \leftarrow 0$
3:      while SHOULDKEEPSAMPLING($T, Samples$) do
4:          $T \leftarrow T + \textsc{GetSample}_P(G_\emptyset, C_\emptyset)$
5:          $Samples \leftarrow Samples + 1$
6:      print $T / Samples$
7:  procedure GETSAMPLE$_P$($G, C$)
8:      $a \leftarrow \textsc{AddNextEvent}_P(G)$
9:      if $a = \bot$ then return $\prod_{e \in G} |C[e]|$
10:     if ISERRONEOUS$_\mathsf{M}$($G$) then exit(“error”)
11:     if $a \in \mathtt{R}$ then
12:         PICKRF$_P$($G, C, a$)
13:     else if $a \in \mathtt{W}$ then
14:         PICKCO$_P$($G, C, a$)
15:         for $r \in G.\mathtt{R}_{\mathtt{loc}(a)} \setminus \mathsf{cprefix}(a)$ do $C[r] \leftarrow C[r] \uplus \{a\}$
16:     else GETSAMPLE$_P$($G, C$)
17: procedure PICKRF$_P$($G, C, a$)
18:     $C[a] \leftarrow \{w \in G \mid \mathsf{consistent}(\mathsf{SetRF}(G, w, a))\}$
19:     randomly choose some $w \in C[a]$
20:     GETSAMPLE$_P$($\mathsf{SetRF}(G, w, a), C$)
21: procedure PICKCO$_P$($G, C, a$)
22:     $C[a] \leftarrow \{w_p \in G \mid \mathsf{consistent}(\mathsf{SetCO}(G, w_p, w))\}$
23:     randomly choose some $w_p \in C[a]$
24:     GETSAMPLE$_P$($\mathsf{SetCO}(G, w_p, a), C$)
```

verification. Similarly, for programs whose state space is completely symmetric, a near-perfect estimate can be achieved with only a few samples.

To deal with such issues, we make ShouldKeepSampling *dynamic*. After a minimum (fixed) number of samples has been taken, ShouldKeepSampling returns **false** if any of the following holds:

- the standard deviation of the current estimated mean M is less than some fixed percentage of M
- the number of executions explored exceeds M
- a maximum (fixed) number of samples has been taken.

While users are free to override and tune the above heuristics, we found them satisfactory in practice, and they did not seem to affect the quality of the produced estimation.

3.5 Evaluation

To evaluate the accuracy of our approach, we estimated the state-space size of various benchmarks, and measured how close the estimation was to the actual

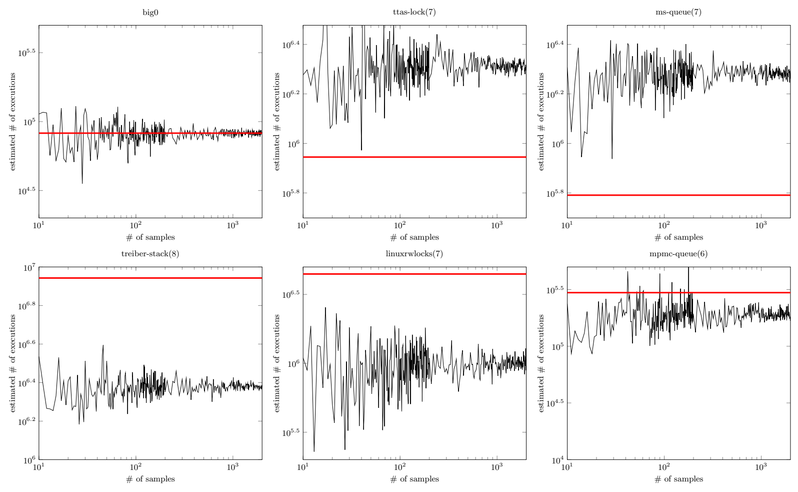

Fig. 4. Estimation accuracy for various benchmarks

```
let sc = po ∪ rf ∪ co ∪ fr
acyclic sc
```

Fig. 5. SC expressed in KAT

size. When selecting tests, we opted for non-symmetric benchmarks (as these are more challenging to estimate), though with a manageable state-space size (so that we can measure how accurate the estimation is). The results we obtained for some representative benchmarks (and a varying sample size) are shown in Fig. 4. The actual state-space size is shown in **red**.

As it can be seen, the state-space estimation is quite useful. It is always within one order of magnitude from the correct number of executions (sometimes under- and sometimes over-approximating the correct value), and converges very quickly: even with only 20 samples we get a fairly accurate estimate of the state space. Finally, estimation is very fast: for instance, it takes about 3 seconds to obtain 2000 samples of `ms-queue`, while verification needs about 1470 seconds.

4 Automatically Porting New MCMs

To port a new MCM into GenMC, one has to define the `corder` relation and implement the consistency checking routine $\mathsf{consistent}_{\mathsf{M}}(G)$ and the MCM-specific error-checking code of $\textsc{IsErroneous}_{\mathsf{M}}(G)$.

Until recently, these routines had to be written manually directly in C++. To some extent, porting new MCMs was automated with Kater [25], a framework for proving metatheoretic properties about MCMs, which can also generate code for checking acyclicity constraints.

```
// Calculation of synchronizes-with
let relseq = [REL] ; ([F] ; po)? ; (rf ; rmw)*
let sw_to_r = relseq ; rf ; [ACQ]
let sw_to_f = relseq ; rf ; po ; [F] ; [ACQ]
let sw = sw_to_r ∪ sw_to_f

// Optimized calculation of happens-before.
// Save the part of 'hb' that does not finish with a reads-from edge
view hb_stable = (po-imm ∪ sw_to_r ; po-imm ∪ sw_to_f)+
let hb = (hb_stable ∪ hb_stable? ; sw_to_r)
assert hb = (po ∪ sw)+

// Coherence : Optimize the checking of irreflexive (hb ; eco)
coherence (hb_stable)

// Sequential consistency order
let eco = (rf ∪ mo ∪ fr)+
let scb = po ∪ rf ∪ mo ∪ fr
let psc = [SC] ; po ; hb ; po ; [SC]
        ∪ [SC] ; ([F] ; hb)? ; scb ; (hb ; [F])? ; [SC]
        ∪ [F∩SC] ; hb ; [F∩SC]
        ∪ [F∩SC] ; hb ; eco ; hb ; [F∩SC]
        ∪ [SC] ; po ; [SC]
acyclic psc

// RC11 error detection
let ww_conflict = [W] ; loc-overlap ; [W]
let wr_conflict = [W] ; loc-overlap ; [R] ∪ [R] ; loc-overlap ; [W]
let conflicting = ww_conflict ∪ wr_conflict
let na_conflict = [NA] ; conflicting ∪ conflicting ; [NA]

// [...]

error RaceNotAtomic   unless  na_conflict ⊆ hb_stable
warning WWRace        unless  [W]; ww_conflict; [W] ⊆ hb_stable
```

Fig. 6. Handling irreflexivity, emptiness, and inclusion constraints

Given an acyclicity constraint of a relation expressible in *Kleene Algebra with Tests* (KAT) [33] (e.g., the `sc` relation of Fig. 5), we can check whether an execution graph G satisfies the acyclicity constraint by taking the product of G with an automaton accepting the (rotational closure of) the language of corresponding to the acyclicity constraint and performing a depth-first search over it. Kater generates code performing such acyclicity checks in linear time in the size of G.

4.1 Beyond Acyclicity Constraints

Although the consistency predicate of SC can be defined as a single acyclicity constraint, more advanced models also check other kinds of constraints, for which Kater could not generate consistency checking code out of the box.

One such model employing different types of constraints is RC11 [35], a fragment of which expressed in KAT can be seen in Fig. 6. It contains three other kinds of constraints, which we discuss below.

First, the `assert` introduces a static constraint about the MCM that is checked at compile-time: it checks that the rewriting of the model that defines

```
Error: Non-atomic race!
Event (2, 4) conflicts with event (1, 3) in graph:
<0, 1> thread_p:
[...]
(1, 4): Wrel (deq.bottom, 1) deque.h:26
[...]
(1, 22): Wrlx (deq.bottom, 4) deque.h:33
[...]
<0, 2> thread_s:
[...]
(2, 3): Racq (deq.bottom, 4) [(1, 22)] deque.h:65
[...]
```

Fig. 7. A GenMC error for a Chase-Lev deque [13]. Events (1, 4) and (2, 3) do not synchronize under C/C++17

`hb` in terms of the `hb_stable` relation is equivalent to the original model which defines `hb` directly. As this constraint can be checked completely statically by Kater, there is no need to generate any code for it.

Second, there is the *coherence* constraint about `hb_stable` dictating that $\texttt{hb\_stable}; (\texttt{rf} \cup \texttt{co} \cup \texttt{fr})^+$ be irreflexive. Such irreflexivity constraints are very common in MCMs and can typically be checked in a very efficient manner. The idea is to represent the `hb_stable` relation using a vector clock. Then, for a read r and a candidate store $s \xrightarrow{\texttt{rf}} r$ in VisitRFs, we check that no other store s' that is `co`-after s is included in r's vector clock[3]. (This would mean that r is aware of a "more recent" store than s.) The total complexity of the generated checks is $\mathcal{O}(N)$ in the size of the graph.

Finally, there are two inclusion constraints introduced by the `error`/`unless` and the `warning`/`unless` constructs. Even though these constraints are not technically part of consistency checking but rather of the IsErroneous$_{\mathsf{M}}$ function, they can still be MCM-specific and so the code checking them has to be mechanically generated.

Generally, for inclusion constraints of the form $a \subseteq b$, we have extended Kater to generate code that calculates the reachable states of a and b, and then checks that the reachable states of a are included in those of b. In the special case where b is represented as a vector clock (as is the case with `hb_stable`), the calculation of b's reachable states is spared, and the generated code only checks whether a's reachable states are contained in b's vector clock.

4.2 Experimenting with MCMs

The above extensions to Kater and GenMC made it possible to fully automate the porting of models like SC [36], RA [34], RC11 [35] and IMM [40]. They also

[3] The procedure for VisitCOs is similar and thus omitted for brevity.

made it easier to experiment with changes in these MCMs, often leading to interesting observations.

One such observation occurred when we “upgraded” GenMC’s RC11 model to use the slightly different `sw` definition of C/C++17 (see Fig. 6), a change that was not supposed to have any impact for most benchmarks. Surprisingly, we found that certain data-structure implementations inadvertently relied on the old definition, and were deemed incorrect (due to improper synchronization) when using the new one (see Fig. 7 for an example). Subtle issues like this underline the need for tools that support automatic porting of MCMs.

5 Performance Improvements

Let us now turn our attention to GenMC’s performance. For a program P with E consistent executions under a model M, the actual verification cost is $E \times C$, where C is the average cost of generating and checking consistency of one program execution. Even if we keep E fixed, engineering optimizations can significantly reduce the cost C and improve the overall performance.

GenMC’s infrastructure is split into two (largely independent) components: the runtime environment, which executes the program under test, and the model checker, which undertakes the construction of consistent execution graphs. The cost per execution, C, can thus be attributed either to the runtime environment (especially in cases where the program code is large or contains some expensive computation) or to the execution graph construction and consistency checking.

In what follows, we address these bottlenecks by presenting two engineering optimizations: one on the runtime environment front (§ 5.1), and one on the model checking front (§ 5.2).

5.1 Reducing Runtime Reliance

Recall that Algorithm 1 repeatedly simulates the execution of the program P to generate all its execution graphs. When the number of shared-memory accesses (which are the ones recorded in an execution graph) are only a small percentage of the program code, the execution time is dominated by code unnecessary for verification.

Obviously, it would be really helpful to reduce reliance on the runtime environment. The key insight in doing so is that we can *cache* the events following a given sequence of read values. To make this concrete, consider the following program where N threads employ a single lock (implemented as a CAS loop) to access a hash table HT:

$$\begin{array}{l} \texttt{while}\ (\neg\texttt{CAS}(lock, 0, 1))\ \ ; \\ HT[i] := \ ... \\ lock := 0 \end{array} \;\Bigg\|\; ... \;\Bigg\|\; \begin{array}{l} \texttt{while}\ (\neg\texttt{CAS}(lock, 0, 1))\ \ ; \\ HT[i] := \ ... \\ lock := 0 \end{array}$$

Observe that the program above has $N!$ (non-blocked) executions, corresponding to all the ways the threads can access the hash table. However, each CAS

Table 1. Performance impact of label caching on data structures

		GenMC/no-cache	GenMC
	Executions	*Time*	*Time*
`treiber-stack(6)`	720	2.26	1.32
`treiber-stack(7)`	5040	57.20	29.37
`treiber-stack(8)`	40 320	1032.09	581.20
`ms-queue(5)`	120	0.91	0.64
`ms-queue(6)`	720	20.98	13.18
`ms-queue(7)`	5040	445.76	294.51
`buf-ring(2)`	20	0.06	0.03
`buf-ring(3)`	1218	0.65	0.48
`buf-ring(4)`	193 280	353.27	253.35
`ttas-lock(5)`	120	0.20	0.14
`ttas-lock(6)`	720	2.47	1.75
`ttas-lock(7)`	5040	62.17	44.24

operation can read one out of two possible values: 0, indicating that the CAS will succeed, the thread will enter its critical section, and release the lock, or 1, indicating that the CAS acquisition failed, and that the thread will try again.

More generally, the value domain used in concurrent programs is small, and it is thus straightforward to record the encountered values (and the corresponding event sequences) for each thread in a trie structure. Then, whenever the model checker e.g., changes a read's `rf`, it gets the sequence of events that will be added after the read (up until the next read) by checking whether the values that the thread read so far have been cached in the trie.

Besides read events, there are other events with read semantics, and thus need to be treated similarly in the trie. One such case are memory allocation events, which are undertaken by GenMC, as opposed to the runtime environment. If the allocated addresses are not recorded in the trie and distributed anew by the model checker, the cache might end up being inconsistent (e.g., if the addresses of later events depend on the allocated address). A better solution is to ensure a deterministic semantics of memory allocation, where the memory address returned depends only the thread identifier and the previous allocations of the same thread.

Caching interacts with the concurrent nature of GenMC itself. GenMC may use multiple worker threads to parallelize the exploration procedure [27], each of which explores a different (unique) execution graph. To avoid contention between worker threads, we equip each GenMC thread with its own (thread-local) cache.

Evaluation As shown in Table 1, caching the encountered events can yield significant performance benefits: depending on the benchmark, GenMC can be 1.4 to 1.8 times faster when caching is employed. In fact, when caching is employed, 90-99% of the calls to the runtime environment are spared for the benchmarks of Table 1, as the respective value sequences are cached.

The reason these savings do not directly translate to runtime gains is that the benchmarks of Table 1 are standard concurrent data structures, which consist almost exclusively of shared-memory accesses. The only gains in these benchmarks stem from the faster access times the trie provides compared to the runtime environment.

5.2 Optimizing Consistency Checking

Runtime aside, most of the remaining time is spent checking consistency of the constructed graph. When it comes to consistency checks, there are two major performance hindrances: (a) the fact that such checks run always, regardless of the program under test and the constructed execution graph, and (b) the consistency checking code itself.

For the first performance issue, we can observe that the nature of the program might render checking full consistency of a graph unnecessary. Let us consider RC11 (Fig. 6) as an example. A large part of the model is devoted to the handling of SC accesses, via the `psc` relation. However, for a program P that does not contain any SC accesses, checking for `psc` acyclicity is unnecessary, and reducing to the Release-Acquire (RA) [34] fragment would suffice. Analogously, if P solely contains SC accesses, we can reduce the verification problem to the SC one.

Leveraging this insight, we employ GenMC with a *model simplification* criterion: if GenMC can statically determine that a particular program P only uses SC/RA accesses, it will try to verify P under SC/RA.

In a somewhat similar manner, we optimize GenMC to try and re-use relations already calculated in an execution graph, rather than recalculating them from scratch. Relations stored in vector clocks (see §4) readily offer such a prime optimization opportunity. When a relation r to be saved in a vector clock is transitive, instead of recalculating it every time we add an event e, we modify GenMC to only calculate the difference between e and its predecessor in r.

For the second performance issue, we observed that that a lot of time was spent allocating memory and indexing into arrays. We therefore optimized the consistency checking code in the following ways.

- To make access to the graph's primitive relations (`rf`, `co`, `po` and `fr`) faster, we rewrote a large part of the existing GenMC infrastructure to use pointers to fetch relation successors/predecessors (instead of array indices), thus sparing one level of indirection.
- To avoid using extra memory when saving relations, we converted all corresponding data structures to intrusive ones, thus significantly reducing access times and memory allocation calls.
- To avoid re-allocating memory for auxiliary data necessary for intermediate computations (e.g., for storing reachable states during inclusion checks), we provided each GenMC thread with a thread-local copy, thereby minimizing memory allocations.

Table 2. Consistency-intensive benchmarks

		Time/SC		Time/TSO		Time/RC11	
	Executions	Old	New	Old	New	Old	New
`szymanski(2)`	78	0.12	0.09	0.14	0.11	0.31	0.17
`szymanski(3)`	1068	2.33	1.36	2.90	1.89	6.90	3.34
`peterson(3)`	588	0.14	0.08	0.16	0.10	0.45	0.18
`peterson(4)`	7360	2.11	0.97	2.51	1.40	7.81	2.80
`parker(1)`	54	0.02	0.02	0.02	0.02	0.04	0.04
`parker(2)`	6701	1.99	1.21	2.51	1.41	6.76	4.13
`dekker_f(3)`	1344	0.51	0.27	0.62	0.39	2.53	0.62
`dekker_f(4)`	26 797	10.39	6.78	15.25	10.23	38.39	16.74
`fib_bench(5)`	218 243	2.14	1.97	3.06	2.45	7.48	4.08
`fib_bench(6)`	2 363 803	23.54	21.38	33.43	26.59	90.57	46.27
`lamport(2)`	16	0.01	0.01	0.01	0.02	0.01	0.02
`lamport(3)`	9216	2.23	1.35	2.67	1.74	7.15	3.26

Evaluation To see how the optimized consistency checks perform, we compared GenMC with its previous version[4]. The results are shown in Table 2. As it can be seen, these performance optimizations greatly improved the performance of GenMC on a set of benchmarks requiring intensive consistency checks.

6 Related Work

Even though there is a plethora of works in stateless model checking [20, 37, 17, 2, 4, 12, 14, 5, 6, 7, 10, 16, 38, 41], few tools can deal with weak MCMs [1, 3, 39, 43, 11, 24]. The only tool among them that supports more than one MCMs is Nidhugg [1], although it is not parametric in the choice of the MCM, and employs a different algorithm for each supported MCM.

As far as other model checking techniques are concerned, SAT/SMT-based bounded model checking (BMC) techniques have been extended to handle weak MCMs [15, 8, 18]. Among these, Dartagnan [18, 22] stands out, as it is also parametric in the choice of the MCM, and porting new MCMs is also automated.

We are not aware of any other (enumerative) model checker that provides an estimation of the state-space size. Even though there are tools that provide a *progress report* (as opposed to a state space estimation), progress reports are typically not as helpful as a size estimation. Java Pathfinder [42] provides a progress bar by assuming that the state space is symmetric (i.e., nodes at the same level in the exploration tree will have same-size subtrees). While the bar grows monotonically, it may not advance at a steady pace, and be ”stuck” at

[4] For this comparison, we disabled the cache (§ 5.1) and the model simplification criterion (§ 5.2), so as to not unfairly penalize the previous GenMC version.

e.g., 99%. DPOR tools like Concuerror [21] and Nidhugg [1] provide a progress report based on how many backtracking options have still to be explored. This number does not monotonically decrease, and is thus of limited use.

7 Summary

In this paper, we enhanced the usability of GenMC (and DPOR in general) by: (a) providing a completion estimate based on a Monte Carlo simulation, and (b) completely automating the porting of new MCMs into the tool. In addition, we improved the tool performance using caching and engineering optimizations. We hope that similar techniques will be leveraged by other researchers and developers working on DPOR tools.

Data-Availability Statement The paper replication package is available at [26]. GenMC is available at [19].

Acknowledgements We thank the anonymous reviewers for their feedback. This work was supported by a European Research Council (ERC) Consolidator Grant for the project "PERSIST" under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: TACAS 2015, LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). DOI: 10.1007/978-3-662-46681-0_28
2. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: POPL 2014, pp. 373–384. ACM, New York, NY, USA (2014). DOI: 10.1145/2535838.2535845
3. Abdulla, P.A., Atig, M.F., Jonsson, B., Leonardsson, C.: Stateless model checking for POWER. In: CAV 2016, LNCS, vol. 9780, pp. 134–156. Springer, Heidelberg (2016). DOI: 10.1007/978-3-319-41540-6_8
4. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proc. ACM Program. Lang. 2(OOPSLA), 135:1–135:29 (2018) DOI: 10.1145/3276505
5. Agarwal, P., Chatterjee, K., Pathak, S., Pavlogiannis, A., Toman, V.: Stateless Model Checking Under a Reads-Value-From Equivalence. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021, pp. 341–366. Springer International Publishing, Cham (2021). DOI: 10.1007/978-3-030-81685-8_16
6. Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P.J.: Context-sensitive dynamic partial order reduction. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017, pp. 526–543. Springer International Publishing, Cham (2017). DOI: 10.1007/978-3-319-63387-9_26
7. Albert, E., Gómez-Zamalloa, M., Isabel, M., Rubio, A.: Constrained dynamic partial order reduction. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, pp. 392–410. Springer International Publishing, Cham (2018). DOI: 10.1007/978-3-319-96142-2_24

8. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: CAV 2013, LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013). DOI: 10.1007/978-3-642-39799-8_9
9. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. 36(2), 7:1–7:74 (2014) DOI: 10.1145/2627752
10. Aronis, S., Jonsson, B., Lång, M., Sagonas, K.: Optimal dynamic partial order reduction with observers. In: TACAS 2018, LNCS, vol. 10806, pp. 229–248. Springer, Heidelberg (2018). DOI: 10.1007/978-3-319-89963-3_14
11. Bui, T.L., Chatterjee, K., Gautam, T., Pavlogiannis, A., Toman, V.: The Reads-from Equivalence for the TSO and PSO Memory Models. Proc. ACM Program. Lang. 5(OOPSLA) (2021) DOI: 10.1145/3485541
12. Chalupa, M., Chatterjee, K., Pavlogiannis, A., Sinha, N., Vaidya, K.: Data-centric dynamic partial order reduction. Proc. ACM Program. Lang. 2(POPL), 31:1–31:30 (2017) DOI: 10.1145/3158119
13. Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In: SPAA 2005, pp. 21–28. ACM (2005). DOI: 10.1145/1073970.1073974
14. Chatterjee, K., Pavlogiannis, A., Toman, V.: Value-Centric Dynamic Partial Order Reduction. Proc. ACM Program. Lang. 3(OOPSLA) (2019) DOI: 10.1145/3360550
15. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS 2004, LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). DOI: 10.1007/978-3-540-24730-2_15
16. Coons, K.E., Musuvathi, M., McKinley, K.S.: Bounded Partial-Order Reduction. In: OOPSLA 2013, pp. 833–848. ACM, Indianapolis, Indiana, USA (2013). DOI: 10.1145/2509136.2509556
17. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL 2005, pp. 110–121. ACM, New York, NY, USA (2005). DOI: 10.1145/1040305.1040315
18. Gavrilenko, N., Ponce-de-León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: Relation analysis for compact SMT encodings. In: Dillig, I., Tasiran, S. (eds.) CAV 2019, pp. 355–365. Springer International Publishing, Cham (2019). DOI: 10.1007/978-3-030-25540-4_19
19. Kokologiannakis, M.: *GenMC: Generic model checking for C programs.* URL: https://github.com/MPI-SWS/genmc
20. Godefroid, P.: Software Model Checking: The VeriSoft Approach. Form. Meth. Syst. Des. 26(2), 77–101 (2005) DOI: 10.1007/s10703-005-1489-x
21. Gotovos, A., Christakis, M., Sagonas, K.: Test-driven development of concurrent programs using concuerror. In: Rikitake, K., Stenman, E. (eds.) Erlang 2022 2011, pp. 51–61. ACM (2011). DOI: 10.1145/2034654.2034664
22. Haas, T., Meyer, R., Ponce de León, H.: CAAT: Consistency as a Theory. Proc. ACM Program. Lang. 6(OOPSLA2) (2022) DOI: 10.1145/3563292
23. Knuth, D.E.: Estimating the Efficiency of Backtrack Programs. Math. Comput. 29(129), 121–136 (1975) DOI: 10.2307/2005469
24. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. 2(POPL), 17:1–17:32 (2017) DOI: 10.1145/3158105
25. Kokologiannakis, M., Lahav, O., Vafeiadis, V.: Kater: Automating Weak Memory Model Metatheory and Consistency Checking. Proc. ACM Program. Lang. 7(POPL) (2023) DOI: 10.1145/3571212

26. Kokologiannakis, M., Majumdar, R., Vafeiadis, V.: *Enhancing GenMC's Usability and Performance (Replication Package)*(2024). URL: https://zenodo.org/doi/10.5281/zenodo.10018135
27. Kokologiannakis, M., Marmanis, I., Gladstein, V., Vafeiadis, V.: Truly stateless, optimal dynamic partial order reduction. Proc. ACM Program. Lang. 6(POPL) (2022) DOI: 10.1145/3498711
28. Kokologiannakis, M., Marmanis, I., Vafeiadis, V.: Unblocking Dynamic Partial Order Reduction. In: CAV 2023, pp. 230–250. Springer (2023). DOI: 10.1007/978-3-031-37706-8_12
29. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: PLDI 2019, ACM, New York, NY, USA (2019). DOI: 10.1145/3314221.3314609
30. Kokologiannakis, M., Ren, X., Vafeiadis, V.: Dynamic Partial Order Reductions for Spinloops. In: FMCAD 2021, pp. 163–172. IEEE (2021). DOI: 10.34727/2021/isbn.978-3-85448-046-4_25
31. Kokologiannakis, M., Vafeiadis, V.: BAM: Efficient Model Checking for Barriers. In: NETYS 2021, LNCS, Springer, Heidelberg (2021). DOI: 10.1007/978-3-030-91014-3_16
32. Kokologiannakis, M., Vafeiadis, V.: GenMC: A model checker for weak memory models. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021, LNCS, vol. 12759, pp. 427–440. Springer, Heidelberg (2021). DOI: 10.1007/978-3-030-81685-8_20
33. Kozen, D.: Kleene Algebra with Tests. ACM Trans. Program. Lang. Syst. 19(3) (1997) DOI: 10.1145/256167.256195
34. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming Release-acquire Consistency. In: POPL 2016, pp. 649–662. ACM, St. Petersburg, FL, USA (2016). DOI: 10.1145/2837614.2837643
35. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI 2017, pp. 618–632. ACM, Barcelona, Spain (2017). DOI: 10.1145/3062341.3062352
36. Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Trans. Computers 28(9), 690–691 (1979) DOI: 10.1109/TC.1979.1675439
37. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing Heisenbugs in concurrent programs. In: OSDI 2008, pp. 267–280. USENIX Association (2008). URL: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf
38. Nguyen, H.T.T., Rodríguez, C., Sousa, M., Coti, C., Petrucci, L.: Quasi-optimal partial order reduction. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, LNCS, vol. 10982, pp. 354–371. Springer, Heidelberg (2018). DOI: 10.1007/978-3-319-96142-2_22
39. Norris, B., Demsky, B.: CDSChecker: Checking concurrent data structures written with C/C++ atomics. In: OOPSLA 2013, pp. 131–150. ACM (2013). DOI: 10.1145/2509136.2509514
40. Podkopaev, A., Lahav, O., Vafeiadis, V.: Bridging the gap between programming languages and hardware weak memory models. Proc. ACM Program. Lang. 3(POPL), 69:1–69:31 (2019) DOI: 10.1145/3290382
41. Rodríguez, C., Sousa, M., Sharma, S., Kroening, D.: Unfolding-based Partial Order Reduction. In: CONCUR 2015, LIPIcs, pp. 456–469. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015). DOI: 10.4230/LIPIcs.CONCUR.2015.456

42. Wang, K., Converse, H., Gligoric, M., Misailovic, S., Khurshid, S.: A Progress Bar for the JPF Search Using Program Executions. ACM SIGSOFT Softw. Eng. Notes 43(4), 55 (2018) DOI: 10.1145/3282517.3282525
43. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: PLDI 2015, pp. 250–259. ACM, New York, NY, USA (2015). DOI: 10.1145/2737924.2737956