HMC: Model Checking for Hardware Memory Models

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

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

Abstract

Stateless Model Checking (SMC) is an effective technique for verifying safety properties of a concurrent program by systematically exploring all of its executions. While SMC has been extended to handle hardware memory models like x86-TSO, it does not adequately support models that allow load buffering behaviours, such as the POWER, ARMv7, ARMv8, and RISC-V models. Existing SMC tools either do not consider such behaviours in the name of efficiency, or do not scale so well due to the extra complexity induced by these behaviours.

We present HMC, the first efficient SMC algorithm that can verify programs under all hardware memory models in a sound, complete, and optimal fashion. We implement HMC in a tool for C programs, and show that it outperforms the state-of-the-art tools that can handle similar memory models. We demonstrate the efficiency of HMC by verifying code currently employed in production.

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:
Michalis Kokologiannakis and Viktor Vafeiadis. 2020. HMC: Model Checking for Hardware Memory Models. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378480

1 Introduction

Stateless Model Checking (SMC) [20, 21, 31] coupled with *Dynamic Partial Order Reduction* (DPOR) [18, 5, 1] is an effective combination for verifying concurrent C/C++ programs [22, 28], which is based on a simple idea: to enumerate all the possible executions of a finite program and check that none of them violates the desired safety property. Its effectiveness, however, is highly dependent on the *memory consistency model* under which verification takes place.

Memory consistency models can be split in two categories depending on whether and how they allow "load buffering" outcomes such as the one shown below:

$$\begin{array}{l||l} a := x \ /\!/\ 1 & b := y \ /\!/\ 1 \\ y := 1 & x := 1 \end{array} \qquad (\textsc{lb})$$

Models in the first category, known in the literature as porf-acyclic models (for reasons that will become apparent in §2), forbid such outcomes. Models in this category include SC [30], TSO [33], PSO [40], and RC11 [29]. Verification under porf-acyclic models has been studied extensively and there exist efficient verification tools that support them (e.g., [1, 32, 15, 26, 4, 27]).

Models in the second category allow the annotated behaviour of LB: they effectively support the reordering of a load after a subsequent store to a different memory location. With the exception of C11 [11], however, they do not allow all such reorderings, since that would allow out-of-thin-air behaviours like the one illustrated below.

$$\begin{array}{l||l} a := x \ /\!/\ 1 & b := y \ /\!/\ 1 \\ \textbf{if } a = 1 \textbf{ then } y := 1 & \textbf{if } b = 1 \textbf{ then } x := 1 \end{array} \qquad (\textsc{lb+ctrl})$$

To rule out the annotated outcome from this second program, models of this second category follow one of the following two approaches.

Some models keep track of dependencies between instructions and restrict the reordering of instructions that are dependent. These *dependency-tracking* models form a broad class that includes all remaining hardware models and a few low-level software models: namely, POWER [9], ARMv7 [9], ARMv8 [35], RISC-V [43], IMM (Intermediate Memory Model) [34], and LKMM (Linux-kernel model) [8].

Other models, such as Promising [25] and Weakestmo [14], distinguish between LB and LB+CTRL in a more complex way without tracking dependencies. They consider multiple program executions to justify the outcomes of a single execution.

In this paper, we focus on the *dependency-tracking* models. Verification under these models is substantially more complex than under the porf-acyclic ones because they allow outcomes (such as that of LB) that cannot be generated by executing the program following the order of its instructions. Thus, existing SMC tools take one of the following two approaches: they either disregard load-buffering behaviours

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378480

and restrict to a fragment of the memory model for the sake of efficiency, e.g., [26, 27], or they sacrifice scalability by naively emulating all the possible out-of-order executions that could arise in a program, e.g., [3, 32, 36].

Of course, neither solution is completely satisfactory. First, disregarding such behaviours could hide bugs, since there is code in production that can exercise similar behaviours. For example, Linux kernel developers write concurrency code for a dependency-tracking model [8], and the correctness of such code is often of critical importance [16]. Second, scalability is of great importance if verification tools are to be used as an actual aid to developers of concurrent code. Naturally, it would be extremely desirable to have a technique that can handle dependency-tracking models in an effective manner.

In this paper, we develop HMC (Hardware-memory Model Checker), the first efficient SMC algorithm that supports dependency-tracking models. In HMC, we do not try to directly emulate out-of-order execution as the prior work [32, 36] does. We instead extend the approach of GenMC [27] to lift its assumption of porf-acyclicity. While executing the program in instruction order, HMC keeps track of dependencies between instructions, which enables it to explore alternative options where the instructions appear to have been executed out of order. Our algorithm, HMC, is largely parametric in the choice of the memory model, and can be instantiated with any of the aforementioned porf-acyclic and dependency-tracking models: SC, TSO, PSO, RC11, POWER, ARMv7, ARMv8, RISC-V, IMM, LKMM.

Our contributions can be summarized as follows:

- (§2) We show how executions can be modelled in weak memory models, as well as how dependencies can be calculated dynamically, in order to be taken into account by the model checker.
- (§3) We present the HMC algorithm and show how it works on the LB program.
- (§4) We implement HMC in a verification tool for C programs with IMM [34] as the memory model.
- (§5) We demonstrate HMC's effectiveness by showing that it outperforms the state-of-the-art SMC tools that can handle similar memory models, and that it is only moderately slower than tools supporting only porf-acyclic models. We also show that our tool can handle real code employed in production.

We conclude with a discussion of related (§6) and future (§7) work.

2 Preliminaries

The goal in stateless model checking is to enumerate all the executions of a given program and check for safety violations. Following the standard declarative (a.k.a. axiomatic) approach of Alglave et al. [9], we represent the executions of a concurrent program as *execution graphs*. In this section, we will present how programs can be mapped to execution graphs for an arbitrary memory model that tracks dependencies, while in the next section we will present how to systematically enumerate the graphs for a given program.

In the remainder of this section, we first define a toy programming language based on LLVM-IR (§ 2.1). Then, we define the structure of execution graphs (§ 2.2), and show how these graphs are constructed from programs (§ 2.3).

Notation Given a relation r, we write $\mathsf{r}^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closure of r, respectively. We write r^{-1} for the inverse of r. Given relations r_1 and r_2, we write $\mathsf{r}_1;\mathsf{r}_2$ for $\{(a,b) \mid \exists c.\ (a,c) \in \mathsf{r}_1 \land (c,b) \in \mathsf{r}_2\}$, i.e., their relational composition.

2.1 Programming Language

In order to show how programs are mapped to graphs, we formulate a simple assembly programming language. Our language is inspired from LLVM-IR, although, in contrast to LLVM-IR, it is untyped. To avoid cluttering the presentation, we omit features such as function calls and indirect jumps, which are orthogonal to the concurrency semantics.

Instructions, $i \in \mathsf{Inst}$, are given by the following grammar:

$$\begin{aligned} i ::= {} & r := e \mid \mathsf{if}\ r\ \mathsf{goto}\ n \mid \mathsf{error} \mid [r]^{o_\mathsf{W}} := r' \mid r := [r']^{o_\mathsf{R}} \mid \\ & \mathsf{fence}^{o_\mathsf{F}} \mid r := \mathsf{FAI}^{o_\mathsf{R},o_\mathsf{W}}(r_1, r_2) \mid r := \mathsf{CAS}^{o_\mathsf{R},o_\mathsf{W}}(r_1, r_2, r_3) \end{aligned}$$

where r ranges over registers, n over integers, and e over simple expressions built from registers, integers, and arithmetic operators:

$$e ::= n \mid r \mid e_1 + e_2 \mid e_1 - e_2 \mid \dots$$

Finally, o_R, o_W, and o_F range over *access modes*, which are used to distinguish different types of memory accesses.[1]

Returning to the instructions, $r := e$ assigns the value of e to register r (without any effect on memory); $\mathsf{if}\ r\ \mathsf{goto}\ n$ jumps to n if r has a non-zero value; error halts the program with an error; $r := [r']^{o_\mathsf{R}}$ reads the value in the address pointed by r' and stores it in register r; $[r]^{o_\mathsf{W}} := r'$ stores the value contained in r' in the address contained in r; $\mathsf{fence}^{o_\mathsf{F}}$ is used to place global barriers; $r := \mathsf{FAI}^{o_\mathsf{R},o_\mathsf{W}}(r_1, r_2)$ (fetch-and-increment) atomically increments the value stored in location r_1 by the value stored in r_2 and returns the old value to r; and $r := \mathsf{CAS}^{o_\mathsf{R},o_\mathsf{W}}(r_1, r_2, r_3)$ (compare-and-swap) atomically compares the value stored in location r_1 with the value of r_2, and if they are equal, it replaces the value in location r_1 with the value stored in r_3.

A sequential program, *sprog*, is simply a collection of instructions (defined as a finite map from $\mathbb{N}$ to instructions), while a concurrent program, P, is a parallel composition of sequential programs (defined as a finite map from thread identifiers to sequential programs). In our examples, we use vertical alignment to denote sequences of instructions and '$\|$' for the parallel composition of threads.

[1] The precise definition of access modes is not important for this paper and depends on the memory model. For example, C11 [11] has non-atomic, relaxed, acquire, release, acquire-release, and SC accesses.

2.2 Memory Models as Sets of Execution Graphs

The runs of a program under a particular memory model are represented as a set of execution graphs satisfying the memory model's consistency predicate.

Each execution graph G contains a set of events representing the program's memory accesses together with additional information about them (e.g., their type, value, and dependencies).

Definition 2.1. An *event*, $e \in \mathsf{Event}$, can be either:

- an initialization event: $\langle \mathtt{init}\ x \rangle$ where $x \in \mathsf{Loc}$ is the location being initialized.
- a thread event: $\langle t, i \rangle$ where $t \in \mathsf{Tid}$ is a thread identifier, and $i \in \mathbb{N}$ is a serial number inside each thread.

The set of all initialization events is denoted by E_0. The functions $\mathtt{tid}$ and $\mathtt{iid}$ return the thread identifier and the serial number of an event, respectively.

Given the above representation of events, we induce the *program order*, which is a partial order on events given by:

$$\mathsf{po} \triangleq \mathsf{E}_0 \times (\mathsf{Event} \setminus \mathsf{E}_0) \cup \left\{ \begin{matrix} \langle \langle t_1, i_1 \rangle, \\ \langle t_2, i_2 \rangle \rangle \end{matrix} \;\middle|\; t_1 = t_2 \wedge i_1 < i_2 \right\}$$

Intuitively, initialization events precede all non-initialization events, while events in the same thread are ordered according to their serial numbers.

Execution graphs map events to labels, which describe the instructions responsible for the given event.

Definition 2.2. A *label*, $l \in \mathsf{Lab}$, takes one of the following forms:

- Read label: $\mathsf{R}^o_s(x)$ where $x \in \mathsf{Loc}$ is the location accessed, $s \in \{\mathtt{not\text{-}ex}, \mathtt{ex}\}$ is an exclusiveness mark (see below), and o records the access mode.
- Write label: $\mathsf{W}^o_s(x, v)$ where $x \in \mathsf{Loc}$ records the location accessed, $v \in \mathsf{Val} \triangleq \mathbb{Z}$ the value written, $s \in \{\mathtt{not\text{-}ex}, \mathtt{ex}\}$, and o records the access mode.
- Fence label: F^o where o records the access mode.
- Error label: $\mathtt{error}$.

Read and write labels include a flag s that marks them as *exclusive*. Exclusive reads and writes are part of a *read-modify-write (RMW)* operation such as FAI or CAS. Exclusive accesses typically appear as adjacent pairs—an exclusive read is immediately followed by a corresponding exclusive write—except for the case of a "failing" CAS operation.

Now, we can formally define an execution as follows:

Definition 2.3. An *execution* G consists of:

1. a sequence $G.\mathsf{TE}$ of thread events. We denote by $G.\mathsf{E}$ the set of all events in G, i.e., including initialization events; that is, $G.\mathsf{E} \triangleq \mathsf{E}_0 \cup G.\mathsf{TE}$.
2. a labelling function $G.\mathtt{lab} : G.\mathsf{TE} \longrightarrow \mathsf{Lab}$ that maps events to their corresponding labels. We extend the labelling function to map initialization events to the corresponding write label, i.e., $G.\mathtt{lab}(\langle \mathtt{init}\ x \rangle) = \mathsf{W}(x)$.

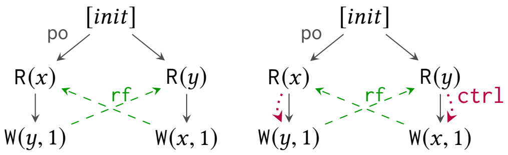


Figure 1. Executions of LB (left) and LB+CTRL (right) corresponding to the annotated outcomes of §1.

$G.\mathtt{lab}$ naturally induces the functions $G.\mathtt{typ}$, $G.\mathtt{mod}$, $G.\mathtt{loc}$, and $G.\mathtt{val}$, that return a label's type, mode, location, and value, respectively. We use $G.\mathsf{R}$, $G.\mathsf{W}$, $G.\mathsf{F}$, and $G.\mathtt{error}$ to denote the set of events of the respective type. We use subscripts to further restrict the event modifiers (e.g., $G.\mathsf{W}_x = \{w \in G.\mathsf{W} \mid G.\mathtt{loc}(w) = x\}$).

3. a function $G.\mathtt{rf} : G.\mathsf{R} \longrightarrow G.\mathsf{W}$, called the *reads-from* function, that maps each read event to the write (of the same location) that determines the read's value.
4. a function $G.\mathtt{addr} : (G.\mathsf{R} \cup G.\mathsf{W}) \longrightarrow \mathcal{P}(G.\mathsf{R})$ that records the *address dependencies* of memory accesses.
5. a function $G.\mathtt{data} : G.\mathsf{W} \longrightarrow \mathcal{P}(G.\mathsf{R})$ that records the *data dependencies* of writes.
6. a function $G.\mathtt{ctrl} : G.\mathsf{TE} \longrightarrow \mathcal{P}(G.\mathsf{R})$ that records the *control dependencies* of events.

We introduce overline notation to convert the reads-from and dependency functions to relations on events:

$$G.\overline{\mathtt{rf}} \triangleq \{\langle G.\mathtt{rf}(r), r \rangle \mid r \in G.\mathsf{R}\}$$

$$G.\overline{\mathsf{X}} \triangleq \{(r, e) \mid r \in G.\mathsf{X}(e)\} \quad \text{for } \mathsf{X} \in \{\mathtt{addr}, \mathtt{data}, \mathtt{ctrl}\}$$

and assume that all dependencies edges are included in the program order; i.e., $G.\overline{\mathsf{X}} \subseteq \mathsf{po}$ for $\mathsf{X} \in \{\mathtt{addr}, \mathtt{data}, \mathtt{ctrl}\}$. Example executions for LB and LB+CTRL can be seen in Fig. 1. Note that, in figures, we omit the overline for relations; for example, $\overline{\mathtt{rf}}$ is drawn simply as $\mathtt{rf}$.

Each memory model, M, defines a consistency predicate on executions, $\mathsf{cons}_{\mathsf{M}}(.)$, denoting its set of permitted executions. For instance, *sequential consistency* (SC) [30] can be defined by making use of a modification order as follows.

Definition 2.4 (Modification order)**.** A relation $\mathtt{mo}$ is a *modification order* for an execution G iff $\mathtt{mo}$ is a strict partial order, $\mathtt{mo} \subseteq \bigcup_{x \in \mathsf{Loc}} G.\mathsf{W}(x) \times G.\mathsf{W}(x)$, and for every location $x \in \mathsf{Loc}$, $\mathtt{mo}$ is total on $G.\mathsf{W}(x)$.

Definition 2.5 (RMW-atomicity)**.** An execution G satisfies *RMW-atomicity* iff there are no two distinct exclusive reads, $r_1, r_2 \in G.\mathsf{R}_{\mathsf{ex}}$, that have corresponding exclusive writes (i.e., $\langle r_i.\mathtt{tid}, r_i.\mathtt{iid} + 1 \rangle \in G.\mathsf{W}_{\mathsf{ex}}$ for $i \in \{1, 2\}$) and read from the same write $G.\mathtt{rf}(r_1) = G.\mathtt{rf}(r_2)$.

Definition 2.6 (SC)**.** G is sequentially consistent, written $\mathsf{cons}_{\mathsf{SC}}(G)$, iff it satisfies RMW-atomicity and there is a modification order $\mathtt{mo}$ for G such that $\mathsf{po} \cup G.\overline{\mathtt{rf}} \cup \mathtt{mo} \cup (G.\overline{\mathtt{rf}}^{-1}; \mathtt{mo})$ is acyclic.

A memory model M is called porf*-acyclic* iff $\mathsf{cons}_\mathsf{M}(G)$ implies that $\mathsf{po} \cup G.\overline{\mathsf{rf}}$ is acyclic. Examples of porf-acyclic models are SC (Def. 2.6), TSO [33], PSO [40], and RC11 [29].

The *prefix* of an event e is the set of all thread events $e' \in G.\mathsf{TE}$ such that $\langle e', e\rangle \in (G.\overline{\mathsf{rf}} \cup G.\overline{\mathsf{addr}} \cup G.\overline{\mathsf{data}} \cup G.\overline{\mathsf{ctrl}})^+$. A model M is *well-prefixed* if no M-consistent execution G contains an event that is in its own prefix. Well-prefixed models include all the porf-acyclic ones, as well as ARMv7 [9], ARMv8 [35], IMM [34], LKMM [8], POWER [9], and RISC-V [43]. By contrast, the original C11 model [11] is not well-prefixed, since it allows the out-of-thin-air behaviour of LB+CTRL (depicted in Fig. 1).

2.3 From Programs to Execution Graphs

Now that we have defined executions, we will show how programs are mapped to sets of executions. We do so by defining the ExecProgram(P, G) procedure (Algorithm 1), which checks that the execution G corresponds to some run of the program P. Later, in §3.1, we will extend this procedure to also generate the execution incrementally.

ExecProgram interprets the program P and checks that the memory accesses generated match those recorded in G. For each thread t of the program (Line 2), it constructs a configuration of the form $\langle a, \Phi, \Delta\rangle$, where a records the last t-event considered, $\Phi : \mathsf{Reg} \to \mathsf{Val}$ is the *register file* that maps registers to values, and $\Delta : \mathsf{Reg} \to \mathcal{P}(\mathsf{Event})$ is the *dependency set* which maps each register to the set of events used to calculate its value. Initially, a is set to $\langle t, 0\rangle$ signalling that no events have yet been checked for that thread, and every register has the value 0 and no dependencies (Line 3).

The register set includes a special register, the *program counter* (pc), that points to the next instruction to be executed. The program counter is incremented by every instruction (Line 5), except for conditional branches where it is set to a specified value when the condition holds.

The interpretation of a thread proceeds in a loop as long as the program counter points to a valid instruction (Line 4). In each loop iteration, ExecInstruction is called to interpret the current instruction (Line 6). Under the usual SMC assumption that programs are loop-free (or equivalently, that its loops are unrolled to some specified depth), the **while** loop is guaranteed to terminate. Finally, when the loop finishes, ExecProgram checks that all events of G pertaining to thread t have been generated (Line 7).

ExecInstruction does a case analysis over the type of the instruction, updating Φ and recording any dependencies in Δ as appropriate. For memory accesses, it calls the Gen helper function, which checks that the next event of the given thread recorded in G is the expected one: it has the supplied label and the supplied address, data, and control dependencies. Whenever a read event a is generated, Gen returns the value read by looking up the value written by the write from which a reads (Line 39).

Algorithm 1 Check that G is an execution of program P

```
 1: procedure ExecProgram($P, G$)
 2:     for $\langle t, sprog\rangle \in P$ do
 3:         $\langle a, \Phi, \Delta\rangle \leftarrow \langle\langle t, 0\rangle, \lambda r.\, 0, \lambda r.\, \emptyset\rangle$
 4:         while $i \leftarrow sprog(\Phi(pc))$ do
 5:             $\Phi(pc) \leftarrow \Phi(pc) + 1$
 6:             ExecInstruction($G, \Phi, \Delta, a, i$)
 7:         assert $a.\mathsf{iid} = |\{e \in G.\mathsf{TE} \mid e.\mathsf{tid} = t\}|$

 8: procedure ExecInstruction($G, \Phi, \Delta, a, i$)
 9:     case $i \equiv r := e$
10:         $\Phi(r) \leftarrow \Phi(e)$ ; $\Delta(r) \leftarrow \Delta(e)$
11:     case $i \equiv \mathsf{if}\ r\ \mathsf{goto}\ n$
12:         if $\Phi(r) \neq 0$ then $\Phi(pc) \leftarrow n$
13:         $\Delta(pc) \leftarrow \Delta(pc) \cup \Delta(r)$
14:     case $i \equiv \mathsf{error}$
15:         Gen($G, a, \mathtt{error}, \emptyset, \emptyset, \Delta(pc)$)
16:     case $i \equiv r := [r']^{o_\mathsf{R}}$
17:         $v \leftarrow$ Gen($G, a, \mathsf{R}^{o_\mathsf{R}}_{\mathsf{not\text{-}ex}}(\Phi(r')), \Delta(r'), \emptyset, \Delta(pc)$)
18:         $\Phi(r)$ ; $\Delta(r) \leftarrow \{a\}$
19:     case $i \equiv [r]^{o_\mathsf{W}} := r'$
20:         Gen($G, a, \mathsf{W}^{o_\mathsf{W}}_{\mathsf{not\text{-}ex}}(\Phi(r), \Phi(r')), \Delta(r), \Delta(r'), \Delta(pc)$)
21:     case $i \equiv \mathsf{fence}^{o_\mathsf{F}}$
22:         Gen($G, a, \mathsf{F}^{o_\mathsf{F}}, \emptyset, \emptyset, \Delta(pc)$)
23:     case $i \equiv r := \mathsf{FAI}^{o_\mathsf{R}, o_\mathsf{W}}(r_1, r_2)$
24:         $v \leftarrow$ Gen($G, a, \mathsf{R}^{o_\mathsf{R}}_{\mathsf{ex}}(\Phi(r_1)), \Delta(r_1), \emptyset, \Delta(pc)$)
25:         $l \leftarrow \mathsf{W}^{o_\mathsf{W}}_{\mathsf{ex}}(\Phi(r_1), \Phi(r_2) + v)$
26:         Gen($G, a, l, \Delta(r_1), \Delta(r_2) \cup \{a\}, \Delta(pc)$)
27:         $\Phi(r) \leftarrow v$ ; $\Delta(r) \leftarrow \{a\}$
28:     case $i \equiv r := \mathsf{CAS}^{o_\mathsf{R}, o_\mathsf{W}}(r_1, r_2, r_3)$
29:         $v \leftarrow$ Gen($G, a, \mathsf{R}^{o_\mathsf{R}}_{\mathsf{ex}}(\Phi(r_1)), \Delta(r_1), \emptyset, \Delta(pc)$)
30:         $\Delta(pc) \leftarrow \Delta(pc) \cup \{a\} \cup \Delta(r_2)$
31:         if $v = \Phi(r_2)$ then
32:             Gen($G, a, \mathsf{W}^{o_\mathsf{W}}_{\mathsf{ex}}(\Phi(r_1), \Phi(r_3)), \Delta(r_1), \Delta(r_3), \Delta(pc)$)
33:         $\Phi(r) \leftarrow v$ ; $\Delta(r) \leftarrow \{a\}$

34: procedure Gen($G, a, \mathtt{lab}, \mathtt{addr}, \mathtt{data}, \mathtt{ctrl}$)
35:     $a.\mathsf{iid} \leftarrow a.\mathsf{iid} + 1$
36:     assert $a \in G.\mathsf{TE}$
37:     for $\mathsf{X} \in \{\mathtt{lab}, \mathtt{addr}, \mathtt{data}, \mathtt{ctrl}\}$ do
38:         assert $G.\mathsf{X}(a) = \mathsf{X}$
39:     if $a \in G.\mathsf{R}$ then return $v \leftarrow G.\mathsf{val}(G.\mathsf{rf}(a))$
```

We define the executions of a program P under a memory model M as the set of all M-consistent executions G generated by P; i.e., ExecProgram(P, G) terminates without assertion violations and $\mathsf{cons}_\mathsf{M}(G)$ holds. For example, notice how ExecProgram would terminate without assertion violations for LB+CTRL and the right graph of Fig. 1, denoting that this graph is indeed an execution of LB+CTRL. That execution, however, is inconsistent under all dependency-tracking models, since it contains a $\overline{\mathtt{ctrl}} \cup \overline{\mathtt{rf}}$ cycle, which breaks the well-prefixedness property.

Finally, a program is deemed *correct* under a memory model M if none of its executions under M contains an erroneous event.

3 Algorithm Description

In this section, we explain the exploration algorithm of HMC. HMC extends the GenMC algorithm [27] to support all well-prefixed memory models (instead of only $\mathtt{porf}$-acyclic ones). HMC "inherits" from GenMC soundness, completeness, and optimality: i.e., it explores every program execution precisely once and does not explore any other executions.

Similarly to GenMC, HMC uses a consistent execution graph to drive the exploration. Starting from an empty graph, HMC repeatedly interprets the program to find the next event to be added to the graph. Whenever a read is added and more than one reads-from options exist, the algorithm proceeds with one of them and pushes the rest to a work set. When a full graph is reached (which corresponds to one execution of the program), the graph is restricted according to an alternative exploration option from the work set, and the restricted graph is used to drive further exploration. HMC terminates when all options in the work set have been explored.

Note that, whenever a read is added to the graph, HMC detects the available places it can read from through the graph, and not through the program. And since not all possible writes may have been added to the graph when a read is added, whenever HMC adds a write, it checks whether any of the existing reads in the graph can be *revisited* and made to read from the newly added write.

The algorithm consists of two main components: (1) the *interpreter*, which executes the program and produces the next event to be added to an execution graph; and (2) the *exploration algorithm*, which repeatedly calls the interpreter to generate every execution of the program exactly once. These two components can be thought of as running in separate threads or as 'coroutines' calling each other.

In the rest of this section, we describe these two components in turn and conclude with an example run of HMC.

3.1 The HMC Program Interpreter

The interpreter defines two routines: ResetInterpreter(P), which resets the interpreter state to the start of program P; and NextEvent(G), which continues interpreting the program from where it had previously stopped until the next event is added to the graph, after which it returns the newly added event. If no further event can be added (i.e., the program has terminated), it returns $\bot$.

The NextEvent procedure is an incremental version of ExecProgram (Algorithm 1), which instead of checking that all the events resulting from the program belong to the execution graph G, it returns the next event to be added to G.

Algorithm 2 Generating events incrementally

```
procedure Gen($G, a, \mathtt{lab}, \mathtt{addr}, \mathtt{data}, \mathtt{ctrl}$)
    $a.\mathtt{iid} \leftarrow a.\mathtt{iid} + 1$
    if $a \notin G.\mathsf{TE}$ then
        $G.\mathsf{TE} \leftarrow G.\mathsf{TE} +\!\!+ a$
        for $\mathtt{X} \in \{\mathtt{lab}, \mathtt{addr}, \mathtt{data}, \mathtt{ctrl}\}$ do
            $G.\mathtt{X}(a) \leftarrow \mathtt{X}$
        produce $a$
    if $a \in G.\mathsf{R}$ then return $v \leftarrow G.\mathtt{val}(G.\mathtt{rf}(a))$
```

Algorithm 3 The HMC exploration algorithm

```
procedure HMC($P, \mathsf{M}$)
    $\langle G, T, \Gamma \rangle \leftarrow \langle \varnothing, \varnothing, \varnothing \rangle$
    VisitOne($P, \mathsf{M}, G, T, \Gamma$)
    while $\langle r, w, G_{\text{ext}} \rangle \leftarrow$ RemoveMax($\Gamma$) do
        Restrict($G, T, r$)
        $G \leftarrow G \cup G_{\text{ext}}$
        $G.\mathtt{rf}(r) \leftarrow w$
        VisitOne($P, \mathsf{M}, G, T, \Gamma$)

procedure VisitOne($P, \mathsf{M}, G, T, \Gamma$)
    ResetInterpreter($P$)
    while $\mathsf{cons}_{\mathsf{M}}(G) \wedge a \leftarrow$ NextEvent($G$) do
        if $a \in G.\mathtt{error}$ then exit("Error found.")
        if $a \in G.\mathsf{R}$ then
            $T \leftarrow T \cup \{a\}$
            choose some $w_0 \in G.\mathsf{W}_{\mathtt{loc}(a)}$
            $G.\mathtt{rf}(a) \leftarrow w_0$
            for $w \in G.\mathsf{W}_{\mathtt{loc}(a)} \setminus \{w_0\}$ do
                push($\Gamma, \langle a, w, \varnothing \rangle$)
        if $a \in G.\mathsf{W}$ then
            for $r \in T \cap \mathsf{R}_{\mathtt{loc}(a)} \setminus \mathsf{prefix}(a)$ do
                push($\Gamma, \langle r, a, G|_{\{a\} \cup \mathsf{prefix}(a)} \rangle$)
```

Technically, this is done by replacing the Gen procedure of Algorithm 1 with that of Algorithm 2. Whenever the event a passed to Gen does not already belong to $G.\mathsf{E}$, the new Gen adds it to G, sets its components accordingly, and returns it with a **produce** statement. The **produce** statement means that NextEvent halts at that point and returns a to its caller. The next time that NextEvent is called (unless the interpreter state is reset in the meantime), NextEvent will continue executing right after the **produce** statement.

3.2 The HMC Exploration Procedure

HMC(P, M) explores all the executions of the program P under the memory model M. To do so, it uses three variables: (1) the execution G that is currently being explored, (2) the *revisit set* T, which records all reads of G whose reads-from edges can be changed by future writes, and (3) the *work set* Γ, which contains alternative exploration options that have to be considered in the form of tuples $\langle r, w, G_{\text{ext}} \rangle$, where r is a read event in G, w is a write event, and G_{ext} contains the prefix of w that needs to be restored (because G may have changed in the meantime).

Starting with G, T, and Γ being empty (Line 2), HMC calls the VisitOne procedure to generate one full execution extending G (Line 3). In the process of doing so, VisitOne can push some alternative exploration options to Γ. Thus, when VisitOne returns, HMC proceeds to explore an alternative option from Γ. Following the standard DPOR approach, alternative options are considered in reverse addition order, so as to minimize storage requirements. That is, HMC selects an unexplored entry $\langle r, w, G_{\text{ext}} \rangle$ from Γ with a maximal read event r, i.e., where the first component of every other entry in Γ is either r or appears before r in $G.\mathtt{TE}$ (Line 4); then, it restricts G and T to contain only the event r and all events added before it (Line 5); adds the G_{ext} events to G (Line 6); updates r's incoming reads-from edge (Line 7); and calls VisitOne to explore the constructed execution further (Line 8).

VisitOne carries most of the weight of the exploration, as it not only generates an execution extending G, but also pushes alternative exploration options encountered along the way to Γ. It starts by resetting the interpreter to the beginning of the program P (Line 10). At each step, it checks that G is M-consistent and calls NextEvent to add an event a to G (Line 11). If there is no event to be added (because the program has terminated), VisitOne returns.

If a denotes an error, this is reported to the user (Line 12).

If a is a read, it is marked as revisitable (Line 14) and its incoming rf edge is calculated. From all the writes to the same memory location, VisitOne picks one (Line 15) for a to read from (Line 16), and pushes the remaining read-from options to Γ for later exploration (Line 18).

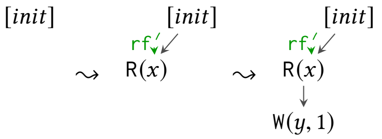
If a is a write, we also have to consider the alternative exploration options in which some revisitable read of G reads from a. Thus, VisitOne iterates over all same-location revisitable reads not in the prefix of a (Line 20). Reads in a's prefix are excluded because revisiting them would yield inconsistent executions with dependency cycles (cf. well-prefixed assumption). For each such read r, the alternative exploration with r reading from a is recorded in the work set Γ (Line 21). The recorded entry also includes G restricted to the events in a's prefix so that they can be restored when the entry is later removed from Γ.

3.3 HMC in Action

Let us now present how our algorithm works with an example. Consider the LB example from §1, the executions of which are depicted in Fig. 2:

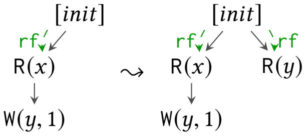
$$\begin{array}{l|l} a := x & b := y \\ y := 1 & x := 1 \end{array} \qquad \text{(LB)}$$

HMC starts from the empty graph and adds events from the leftmost thread to the rightmost one. The first event to be added is a $\mathtt{R}(x)$ event corresponding to $a := x$. Since only the initializing write is present when the read event is added, it will read the initial value. Afterwards, a write event corresponding to $y := 1$ is added, as can be seen below:

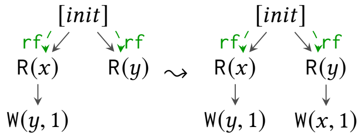


When $\mathtt{W}(y, 1)$ is added, there are no reads of y in the graph, so the for-loop on Line 20 does not execute, and nothing is pushed to the work set.

Next, HMC will add a read event corresponding to $b := y$. Since this read can read both from the initializing write and from $\mathtt{W}(y, 1)$, the algorithm will proceed with, e.g., the first option, and will push the alternative to the work set.



After that, the $\mathtt{W}(x, 1)$ event is added, corresponding to $x := 1$. Since $\mathtt{R}(x)$ is already in the graph and is not in the dependency prefix of $\mathtt{W}(x, 1)$, according to Lines 20 and 21, HMC pushes an alternative exploration where $\mathtt{R}(x)$ reads from $\mathtt{W}(x, 1)$ to the work set. Note that, because $\mathtt{W}(x, 1)$ does not have any dependencies on other events, the prefix pushed to the work set contains only $\mathtt{W}(x, 1)$ itself. Finally, since there are no more events to be added, the first execution is complete (1 from Fig. 2).



At this point, HMC selects alternative reads-from option in reverse addition order (Line 4 of HMC). It therefore explores the alternative for $\mathtt{R}(y)$ reading from $\mathtt{W}(y, 1)$. Then, $\mathtt{W}(x, 1)$ is added (as shown below), but this time it will not revisit the $\mathtt{R}(x)$ again because this is already recorded in Γ. This concludes the exploration of execution 2 from Fig. 2.

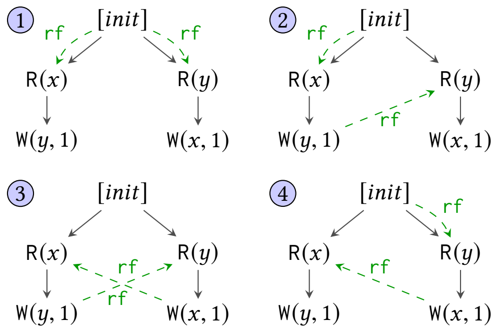
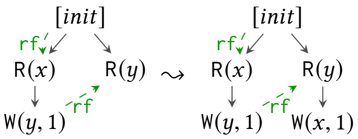
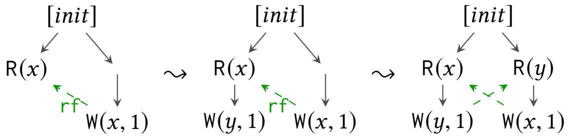


Figure 2. Execution graphs of LB.



Selecting the next entry from Γ, HMC revisits $\mathtt{R}(x)$ and examines the case where it reads from $\mathtt{W}(x, 1)$. In this case, the only remaining events in the graph will be $\mathtt{R}(x)$ and $\mathtt{W}(x, 1)$, since the prefix of $\mathtt{W}(x, 1)$ restored on Line 6 only contained the write itself.

After restriction, $\mathtt{W}(y, 1)$ from the first thread will be added again, followed by $\mathtt{R}(y)$ from the second thread. (Note that the latter is added out-of-order by the algorithm.) Since there are two places for $\mathtt{R}(y)$ to read from, HMC will continue with one of them, e.g., $\mathtt{W}(y, 1)$, and the other will be pushed to Γ. This concludes the exploration of the third execution (③).



Finally, the alternative where $\mathtt{R}(y)$ reads from the initializing write will be explored. In that case, the graph after restriction will be the same as the one in which $\mathtt{R}(y)$ was added during the exploration of the third execution (see the middle graph above). Since all other events were added before $\mathtt{R}(y)$, only the rf edge of that read will differ, thus immediately yielding the fourth execution (④ from Fig. 2).

4 Implementation

The HMC algorithm can be implemented at the level of the assembly language for any particular architecture. For simplicity, we have only implemented HMC for IMM [34], which is a superset of the POWER, ARMv8, and RISC-V memory models, and we used the LLVM Intermediate Representation (LLVM-IR) as our ISA, which can be thought as a platform-independent assembly language.

Our HMC implementation accepts concurrent C programs that use the C11 concurrency primitives and/or the `pthread` library. It comprises: (1) a front-end that uses the `clang` compiler to translate the C code into LLVM-IR, (2) an interpreter for the LLVM-IR based on the interpreter `lli` distributed along with LLVM, (3) a driver running the main HMC algorithm, and (4) code for reporting errors in a format useful for debugging the input program.

In order to keep the interpreter and the driver as separate as possible, we have split the work between these two components somewhat differently from what is shown in §3: the interpreter does not perform any operations on execution graphs, and instead delegates these tasks to the driver.

In more detail, the top-level algorithm (procedure HMC) is implemented by the driver as shown in Algorithm 3 except that the call to VisitOne is implemented just as a call to the beginning of the interpreter (cf. ResetInterpreter). The interpreter then proceeds through the program normally and calls back the driver whenever it encounters a memory access, i.e., whenever ExecInstruction calls Gen. At this point the driver resumes control, and if the event to be added to G does not already belong to G, it executes one iteration of the loop of the VisitOne procedure—i.e., until the next call to NextEvent. Then, the call to the driver returns to the interpreter so that the latter can resume its work and find the next event to be added. Finally, when the interpreter finishes executing the program, it returns from its initial call back to the driver, which pops an entry from Γ, updates the execution graph appropriately (Algorithm 3, Lines 5 to 7), and calls the interpreter again.

The implementation moreover contains a number of optimizations over the algorithm presented in §3. For instance, to avoid having to insert labels in the middle of some container (modelling the sequences), we use vectors with empty labels; these labels denote the places that will be filled out of order, thus achieving amortized $O(1)$ insertion time. Another optimization example is that, instead of saving the whole prefix of an event (Algorithm 3, Line 21), we only keep the part of the prefix that was added after the read being revisited, because the remainder will still be in the graph when the entry is removed from the work set Γ. Finally, in order to store the dependencies of each event we use vector clocks paired with "hole" sets; the vector clock denotes the set of events that an event depends on, while the hole set maintains the exceptions captured in the clock, i.e., events included in the clock that the current one does not depend on.

5 Evaluation

In this section, we evaluate HMC against other state-of-the-art model checking tools. Our evaluation revolves around the following points:

1. HMC is only moderately slower than GenMC, which supports only porf-acyclic models.
2. HMC outperforms all other SMC tools that support non-porf-acyclic memory models—i.e., Nidhugg [3] and CDSChecker [32]—as well as memory model simulators—i.e., Herd [9] and rmem [36].
3. Similar to other SMC tools, HMC cannot be fairly compared against the state-of-the-art SMT-based model checkers. HMC tends to be faster on benchmarks with relatively few executions (i.e., polynomial in the size of the benchmark), whereas SMT-based tools are typically better on parametric benchmarks with an exponential number of executions.

We start by discussing the other model checkers that we compare HMC against.

SMC Tools Nidhugg [1, 3] is a stateless model checker for C programs that supports SC, TSO, PSO, and also offers limited support for POWER and ARMv7. Here, we compared against Nidhugg-power[2]. In general, Nidhugg is not optimal under POWER, and uses a finer equivalence partitioning compared to HMC, which means it can explore exponentially more executions compared to HMC (see § 5.2).

CDSChecker [32] is a stateless model checker for C programs that supports a certain strengthening of the (original) C11 memory model [11] that forbids "out of thin air" outcomes, such as the weak outcome of lb+ctrl. Unlike HMC, it does not track dependencies, but rather uses a notion of promises to support load-store reorderings. This often leads to *infeasible* explorations in programs with C11 'relaxed' memory accesses. In addition, although CDSChecker uses a similarly coarse equivalence partitioning as HMC does, it is non-optimal with respect to its partitioning, which leads to *duplicate* explorations.

Although CDSChecker operates under a non-dependency-tracking memory model, we included it in our tests for two reasons. First, because it is one of the few stateless model checkers that support load-store reorderings, and second, because for most of the benchmarks we used in the paper, the number of consistent executions is the same under C11 and IMM. Thus, the difference in the verification time only reflects the (in)optimality and the efficiency of each tool.

Herd [9] is a memory model simulator that allows users to experiment with different axiomatic memory models on small "litmus test" programs written in a toy language. It supports a wide range of models, but explores executions in a naive fashion and scales rather poorly. For our tests, we compared against Herd-ARMv8.

rmem [37] is a memory model simulator that, among others, supports operational definitions of ARMv8 and RISC-V. Pulte et al. [36] claim that rmem's current operational definitions are suitable for model checking, as they are much faster than the previous ones [35], and tools like Herd. rmem operates on the ARMv8/RISC-V ISA, but does not support dynamic thread creation. Also, unlike SMC tools, rmem does not employ any POR techniques, and so does not scale very well. For our tests, we ran rmem under ARMv8.

SMT-based Model Checkers We also compare HMC with the (SMT-based) bounded model checking (BMC) approach even though SMC and BMC are very different and cannot be fairly compared with one another. Whereas SMC explores all program executions, BMC requires the programs to be given a safety specification and tries to explore only the part of the program pertinent to that specification. BMC tools encode the memory model together with the program's semantics and the specification into a SAT/SMT formula, and query an external solver to determine whether the specification is met. To encode the program, BMC tools require an *a priori* loop bound for programs with loops. By contrast, SMC tools only require that all loops terminate but do not need to unroll them. Since exploration is handled by a SAT/SMT solver, BMC is typically more efficient on benchmarks with an exponential search space, and slower otherwise.

Among the BMC tools, the only one fully supporting dependency-tracking models is Dartagnan [19]. Similarly to HMC, it is parametric in the choice of the memory model and supports the ARMv7, ARMv8, POWER, and LKMM memory models. It works for programs written in a toy language similar—but not identical—to that of Herd. We ran Dartagnan under ARMv7 because we found that, for many of our tests, it is much faster than under ARMv8. We report the difference in running time when Dartagnan is run under ARMv7 and ARMv8 where appropriate.

Benchmark Selection We collected benchmarks from the test suites of the respective tools and used them in our comparisons. The evaluation is done in three steps.

First (§ 5.1), to evaluate the overhead induced by HMC over GenMC, we use the entire benchmark suite of GenMC.

Then (§ 5.2), we use synthetic benchmarks to examine different aspects of the tools and demonstrate how they affect the performance of each tool.

Finally (§ 5.3), we compare HMC with rmem and CDSChecker on the most computationally intensive "real-world" benchmarks from the suites of the respective tools. As can be seen, HMC is significantly faster and can verify implementations of complex concurrent algorithms in a few seconds.

Input Formats Due to the differences between the input languages of the tools under comparison, often a non-trivial amount of work is required to convert programs from one input format to another (e.g., manually unrolling loops, or using fences to simulate C11 accesses). In the larger test cases, conversion to the input formats of Nidhugg, CDSChecker, Herd, and Dartagnan is not even possible because those tools do not support features used by the benchmarks: Nidhugg does not support RMWs, CDSChecker does not support C `struct` types with atomic pointer fields, while Herd and Dartagnan do not support all arithmetic, branching and logical operations.

Experimental Setup We conducted all experiments on a Dell PowerEdge M620 blade system, with two Intel Xeon E5-2667 v2 CPUs (8 cores @ 3.3 GHz) and 256GB of RAM, running a custom Debian-based distribution. We used LLVM 7.0.1 for HMC, GenMC, and Nidhugg (v0.3), commit #da671f7 for CDSChecker, v7.51 for Herd, commit #a6cfaa4 for rmem, and commit #53081dc for Dartagnan with Z3-4.3.2. All reported times are in seconds, unless explicitly noted otherwise. We set the timeout limit to 30 minutes.

5.1 Overhead of HMC

To measure the overhead of HMC over GenMC, we ran both tools on GenMC's test suite (containing over 200 synthetic and non-synthetic tests) along with the benchmarks used in § 5.2 and § 5.3, excluding those that terminate instantly

[2] Nidhugg-arm behaves similarly to Nidhugg-power.

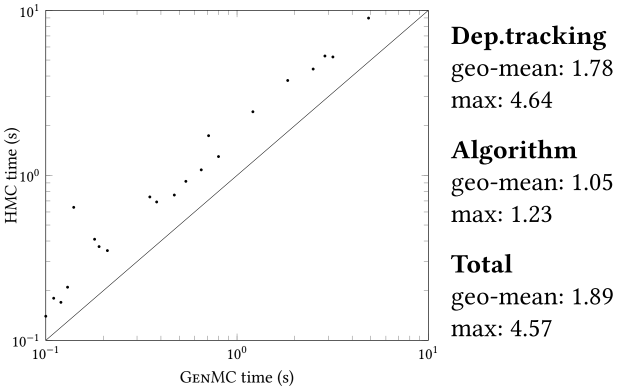


Figure 3. Overhead of HMC over GenMC.

(< 0.1s). The results are shown in Fig. 3. As can be seen, HMC has a standard overhead over GenMC, which we consider acceptable given the coverage of more behaviours.

However, we note that this overhead is not merely due to the increased complexity of the new algorithm and the consistency checks, but rather it is largely due to the naive dynamic calculation of dependencies. To measure the overhead of calculating dependencies, we also replaced the interpreter of GenMC with the dependency-tracking one. Of the average 89% overhead of HMC over GenMC, 78% is due to the calculation of dependencies. The algorithm itself has only 5% average (23% max) overhead over the instrumented version of GenMC that needlessly calculates dependencies.

5.2 Synthetic Benchmarks

We move on to a comparison that highlights the differences among the different model checkers (cf. Table 1).

For the readers(N) benchmark, HMC and CDSChecker perform much better than the other stateless model checkers even though all tools explore the same number of executions (2^N). Herd and rmem do not scale so well for $N = 15$, while Nidhugg times out. For Herd this is because it follows a very naive approach and uses inefficient data structures (e.g., linked lists), while for rmem and Nidhugg this is because, for every event that they add to the constructed trace, they have to check which of its possible parameters are consistent according to the memory model. As can be seen, this is a costly procedure that dominates the running time. We also note that Dartagnan performs very well in this benchmark. This is because the SMT solver is able to come up with an answer really fast without having to explore the exponential number of executions.

Similarly, for nwrites-loc(N), HMC and CDSChecker outperform the other stateless model checkers. In this case, however, this is due to their underlying DPOR algorithms. More specifically, while rmem, Herd and Nidhugg totally order all writes performed by the threads, HMC and CDSChecker leverage the fact that no reader reads from these writes and so avoid ordering them. Thus, rmem, Herd and Nidhugg explore $N!$ executions, while HMC and CDSChecker explore only one execution.

Table 1. Synthetic benchmarks with only loads and stores

	GenMC	HMC	Nidhugg	CDSChecker	Herd	rmem	Dartagnan
readers(5)	0.02	0.02	3.98	0.01	0.02	0.30	0.39
readers(10)	0.11	0.19	818.85	0.19	0.49	3.30	0.40
readers(15)	3.34	6.81	⏲	9.71	25.48	219.25	0.40
nwrites-loc(5)	0.01	0.01	0.20	0.02	0.03	0.71	0.38
nwrites-loc(10)	0.01	0.01	⏲	0.02	666.98	⏲	0.42
nwrites-loc(15)	0.01	0.01	⏲	0.02	⏲	⏲	0.55
nwrites(5)	0.01	0.01	0.21	0.01	0.02	0.60	0.36
nwrites(10)	0.01	0.01	0.69	0.01	0.03	⏲	0.37
nwrites(15)	0.01	0.01	2.61	0.02	0.03	⏲	0.47
mp(10)	0.01	0.02	0.20	0.01	0.14	0.36	0.69
mp(50)	0.02	0.02	1.55	0.02	⏲	0.90	18.02
mp(100)	0.02	0.04	15.31	0.03	⏲	2.26	693.09

`readers(N)`: A thread writing and N threads reading the same memory location (adapted from [2]).

`nwrites-loc(N)`: N threads writing the same memory location (adapted from [4]).

`nwrites(N)`: N threads writing different memory locations.

`mp(L)`: Two threads showcasing the Message Passing idiom, with the reader reading the first variable in a loop until the value true has been read. L is the loop bound.

Concluding Table 1, the nwrites(N) and mp(L) benchmarks demonstrate interesting aspects of rmem and Dartagnan, respectively. For nwrites(N), while all other SMC tools explore only one execution by observing that the threads write to different memory locations, rmem explores $N!$ executions because it imposes a single total ordering across the writes of *all* memory locations. For mp(L), while the number of executions explored by all SMC tools increases linearly with L, Dartagnan needs exponential time to verify the program. Moreover, this time exceeds our timeout limit if we run Dartagnan under ARMv8 (as opposed to ARMv7).

Next, we move on to Table 2, which contains somewhat more challenging benchmarks, where Herd consistently times out. While the observations for Nidhugg and rmem are similar to those for Table 1 above, this table provides us with some useful insight regarding the differences between CDSChecker and HMC, but also between Dartagnan and stateless model checkers in general.

For the first two benchmarks, both HMC and CDSChecker outperform Dartagnan by a large factor. Furthermore, if we run Dartagnan under ARMv8 for parker(N), then the tool times out, even for $N = 3$. Indeed, in this case our observations agree with those of Gavrilenko et al. [19]: when the executions do not grow exponentially as the loop bound gets larger, stateless model checking tools are much faster than bounded model checkers. This is not only because the SMT solver has a worst case exponential complexity, but also because even the encoding fed to the SMT solver may be larger than the state space of the program.

In the last two benchmarks, the situation is reversed: Dartagnan outperforms both CDSChecker and HMC. As

Table 2. Synthetic benchmarks taken from SV-COMP [41]

	GenMC	HMC	Nidhugg	CDSChecker	Herd	rmem	Dartagnan
peterson(10)	0.03	0.05	2.84	0.05	🕒	5.90	9.06
peterson(20)	0.06	0.12	28.08	0.17	🕒	31.78	49.57
peterson(30)	0.17	0.39	137.19	0.40	🕒	95.47	169.79
parker(3)	0.04	0.07	42.85	0.28	🕒	78.84	2.21
parker(4)	0.09	0.18	172.37	0.82	🕒	231.95	4.46
parker(5)	0.14	0.31	536.35	1.95	🕒	504.02	9.64
szymanski(1)	0.05	0.07	0.64	0.45	🕒	1.32	0.66
szymanski(2)	82.04	143.86	110.01	🕒	🕒	101.54	1.31
szymanski(3)	🕒	🕒	🕒	🕒	🕒	🕒	2.89
lamport(2)	0.02	0.02	0.76	0.02	🕒	11.01	2.29
lamport(3)	1.26	2.55	🕒	24.41	🕒	🕒	5.54
lamport(4)	🕒	🕒	🕒	🕒	🕒	🕒	15.96

peterson(L): Two threads performing Peterson's mutual-exclusion algorithm, with L being the loop bound.

parker(L): A recreation of a bug in JDK, adapted from [1]. L is the loop bound.

szymanski(N): Two threads perform Szymanski's mutual-exclusion algorithm N times.

lamport(N): N threads perform Lamport's fast mutex algorithm.

explained before, this is because the number of executions grows exponentially, much faster compared to the encoding fed to the SMT solver by Dartagnan. As a side note, we mention that Dartagnan under ARMv8 needs 67 and 484 seconds for szymanski(3) and lamport(4) respectively.

Among our benchmarks, szymanski(N) and peterson(L) are the only cases where HMC is outperformed by Nidhugg and matched in running time by CDSChecker, respectively, disregarding cases where HMC and CDSChecker both finish almost instantly.

In the case of szymanski(N), the reason for that is the test contains full memory barriers, which both HMC and GenMC treat as acquire-release fences as part of an optimization. (Both tools check for full consistency if an assertion is violated.) In rare cases, this treatment increases the number of consistent executions GenMC/HMC has to explore (in this particular case by four orders of magnitude), resulting in an increased running time. Of course, in tests where both Nidhugg and HMC explore the same number of executions, HMC outperforms Nidhugg by a very large factor anyway.

In the case of peterson(L), the reason for that is that HMC dynamically calculates dependencies between instructions (as explained in Section 2 and 3), which makes it slower for large loop bounds. CDSChecker, on the other hand, simply executes the binary file without calculating dependencies, and has thus minimal overhead. To support load-store reorderings, it uses a notion of promises, but since peterson(L) contains no load-store pairs, and has only acquire/release accesses, CDSChecker does not explore any promises, and terminates quickly. In the case of szymanski(N), however, CDSChecker's notion of promises make it explore many infeasible executions, which cripple its performance. In all other benchmarks, HMC outperforms CDSChecker by a large factor, mostly because CDSChecker explores duplicate executions.

Table 3. Synthetic benchmarks with RMW instructions

	GenMC	HMC	CDSChecker	Herd	rmem	Dartagnan
ainc(3)	0.01	0.01	0.03	0.06	0.35	0.44
ainc(4)	0.01	0.01	0.27	1.20	1.07	0.44
ainc(5)	0.02	0.02	20.39	74.98	6.79	0.53
binc(3)	0.01	0.02	0.18	10.99	43.84	0.47
binc(4)	0.03	0.04	100.27	🕒	🕒	0.52
binc(5)	0.43	0.92	🕒	🕒	🕒	0.59
indexer(12)	0.02	0.03	0.58		🕒	
indexer(13)	0.06	0.09	79.50		🕒	
indexer(14)	0.36	0.62	🕒		🕒	
indexer(15)	2.48	4.54	🕒		🕒	

ainc(N): N threads perform $\mathsf{FAI}(x)_{\texttt{rlx}}$.

binc(N): N threads perform $\mathsf{FAI}(x)_{\texttt{rlx}}; \mathsf{FAI}(y)_{\texttt{rlx}}$.

indexer(N): A classic benchmark by Flanagan and Godefroid [18] designed to demonstrate the benefits of DPOR compared to classic POR techniques. N threads and each thread adds four entries in a shared hash tables. Collisions occur for $N \geq 12$.

Finally, in Table 3, we present some synthetic benchmarks that contain RMW accesses. We exclude Nidhugg from that table because it does not support RMW accesses under POWER and ARMv7. As can be seen, HMC scales fairly well for these benchmarks.

CDSChecker, by contrast, scales poorly because it explores a very large number of infeasible executions. In some of the benchmarks, they are even four orders of magnitude more than the consistent ones. Similarly to to szymanski(N) in the previous table, infeasible executions arise due to the way CDSChecker handles porf cycles and release sequences. The problem manifests itself especially when relaxed accesses are involved. When, for example, we change all accesses of ainc(5) to acquire/release accesses, CDSChecker terminates in 0.07 seconds, and explores much fewer infeasible executions.

Dartagnan, on the other hand, scales nicely both for the ainc(N) and binc(N) benchmarks. The SMT solver manages to establish the supplied assertion without exploring the entire exponential search space, presumably by leveraging the fact that addition is commutative.

For the indexer(N) benchmark, we had to exclude Herd and Dartagnan from the table because their input language does not support all the necessary constructs (e.g., multiplication operations or proper conditional branching). In addition, rmem times out even for $N = 12$, presumably because of the many accesses to different memory locations.

5.3 "Real World" Benchmarks

We proceed by testing HMC on more realistic workloads. For the rest of this section, we exclude Nidhugg, Herd and Dartagnan from our comparisons. Nidhugg does not handle RMW instructions (which are required for these benchmarks), while Herd and Dartagnan operate on a toy language that is insufficient for these benchmarks, and they also require a complete rewrite of the code in their input format, which is tedious and very restricting for larger benchmarks

Table 4. Benchmarks adapted from Pulte et al. [36]

	GenMC	HMC	rmem
DQ/211-2-1	0.11	0.17	172.34
DQ-opt/211-2-1	0.11	0.17	770.45
STC/210-011-000	0.06	0.06	1100.35
STC-opt/210-011-000	0.07	0.10	1154.68
QU/100-100-010	0.05	0.06	1099.20
QU-opt/100-100-010	0.04	0.06	⏲

DQ: An implementation of the Chase-Lev deque.
STC: An implementation of the Treiber stack.
QU: An implementation of the Michael-Scott queue.

(e.g., no support for loops). Thus, we only compare against rmem, CDSChecker, and GenMC.

That said, since rmem and CDSChecker require a different format as input, we compare against each tool on the benchmarks used in the respective paper. Unfortunately, we cannot use CDSChecker on rmem's benchmarks since they contain structs with atomic pointer fields (which are not supported by CDSChecker), nor can we use rmem on CDSChecker's benchmarks, since these benchmarks are too large to be translated to rmem's input language.

We begin by comparing against the most intensive benchmarks for which Pulte et al. [36] published results for rmem (cf. Table 4). While rmem operates on litmus tests, Pulte et al. [36] provide a C++ version of these benchmarks, which we converted to C. We note that the "opt" version of each benchmark differs from the non-opt version in that it has "less" synchronization, i.e., the opt version uses weaker access modes for some accesses.

As can be seen, GenMC and HMC outperform rmem by a very large margin, which is consistent with the outcomes of § 5.2. Since rmem does not leverage POR techniques, it embarks on many redundant explorations. In addition, the time for GenMC and HMC does not change much for the variations of each benchmark; by contrast, the time for rmem changes dramatically, even for DQ, where the number of consistent executions (according to GenMC's equivalence partitioning) remains the same across the opt and the non-opt version.

We continue with the most intensive benchmarks published by Norris and Demsky [32] for CDSChecker, which are shown in Table 5.

Before going into details though, we mention that the two tools use different mechanisms to handle infinite loops. HMC uses a combination of loop bounding along with the transformation of infinite loops with no side-effects into `assume()` statements. CDSChecker, on the other hand, uses a combination of control over the memory liveness and a CHESS-like yield-based fairness system [31]. As we will see, these different mechanisms force the two tools to explore a different number of executions in tests with infinite loops.

Table 5. Benchmarks adapted from Norris and Demsky [32]

	GenMC	HMC	CDSChecker
linuxrwlocks	4.87	8.98	12.23
linuxrwlocks-bnd	0.47	0.76	⏲
mpmc-queue	0.38	0.69	8.85
mpmc-queue-bnd	1.84	3.76	18.24

linuxrwlocks: A reader-writer lock ported from the Linux kernel. Three threads use the lock to read and/or write a shared variable.
mpmc-queue: A multiple-producer multiple-consumer queue. Two threads are enqueueing and then dequeueing an item.

For linuxrwlocks, both GenMC and HMC outperform CDSChecker. That said, they also explore a different number of executions compared to CDSChecker. More specifically, this test case contains infinite loops that GenMC and HMC manage to transform into `assume()` statements, while CDSChecker terminates only if we use an upper bound on the times a thread is allowed to see the same value. Naturally, CDSChecker is sensitive to that upper bound, and as this grows larger the verification becomes even slower. In addition, CDSChecker explores more than 40 times more infeasible executions than consistent executions, presumably due to its handling of relaxed accesses.

Given the above, in an effort to alleviate these discrepancies between the explored executions and perform a more precise comparison, we also manually bounded the test case, thus rendering the mechanisms of all tools that handle infinite loops useless. We also simplified the client code (the threads perform less operations on the lock), since the manual bounding increases the state space of the program.

As can be seen in the respective entry for the first benchmark (linuxrwlocks-bnd), although the test case is simplified, CDSChecker times out, while both GenMC and HMC finish almost instantly. While this may be surprising at a first glance, it is again due to the way CDSChecker handles relaxed accesses and release sequences. More specifically, the definition CDSChecker uses for release sequences leads the tool to explore more executions compared to GenMC, and the relaxed accesses lead the tool to also explore many infeasible executions (plus a few duplicates).

Interestingly enough, however, even though CDSChecker does not verify linuxrwlocks-bnd after 8 hours, if all accesses are changed into acquire/release accesses, it manages to verify it in only 4 seconds. This fact highlights the importance of tracking dependencies and how it can affect the optimality and performance of a tool.

For mpmc-queue, the observations are similar to linuxrwlocks. For the original version of the test case, HMC outperforms CDSChecker, as it transforms infinite loops into `assume()` statements, while CDSChecker is sensitive to the liveness bound. For the bounded version, CDSChecker explores a few more consistent executions (due to its definition of release sequences) and many infeasible ones.

6 Related Work

Stateless Model Checking There are many SMC tools that support reasoning under porf-acyclic memory models such as SC [30], TSO [33], PSO [40], and RC11 [29]; e.g., [20, 31, 18, 5, 26, 4, 45, 1, 15, 10, 6, 27, 38, 23, 24].

Among them, the most closely related tool to our work is GenMC [27], a state-of-the-art model checker for concurrent C programs that is parametric in the choice of an axiomatic memory model, provided that the latter satisfies four basic requirements: porf-acyclicity, extensibility, prefix-closedness, and well-blocking. Our paper weakens the first requirement of GenMC, thereby enabling support for dependency-tracking axiomatic memory models, while preserving parametricity over the precise memory model definition. For this extension, the HMC algorithm, unlike GenMC, records dependencies between instructions, and simulates out-of-order execution, as explained in §3.3 and §4. As shown in §5.1, this imposes a certain overhead over GenMC.

The only SMC tools that support non-porf-acyclic memory models are Nidhugg [3] and CDSChecker [32], which we have discussed in §5.

Memory Model Simulators Apart from SMC, there are a few memory model simulators [9, 11, 39, 37], which can also enumerate all possible executions of a program under a given memory model. That said, they do so in a fairly naive fashion and do not scale beyond small examples. Nevertheless, they are useful in experimenting with different memory models, and Herd [9], in particular, is being used by Linux kernel developers in this way. In §5, we considered the two most prominent such tools: Herd [9] and rmem [37].

Other Approaches Bounded Model Checking (BMC) approaches [17, 7, 13, 19] that handle hardware memory models have also been developed. Among these, the only tool that (similarly to HMC) can handle arbitrary dependency-tracking hardware memory models is Dartagnan, which we have discussed in §5. In general, BMC approaches do not scale so well as the size of the program grows, and consume much more memory [1, 28]. On the other hand, they support data non-determinism, which is not supported by stateless model checking tools.

There have also been developed frameworks like [44, 12] designed to help reasoning about memory model consistency and specifications. Such tools can also be used for the verification of small litmus tests under dependency-tracking models, however, as in the case of memory model simulators, they do not scale to larger programs.

Finally, TriCheck [42] is a tool that can verify that both the mapping of small litmus tests to an ISA, as well as the architectural implementation of said ISA, satisfy a memory-model consistency predicate. This approach is orthogonal to ours, since we take correctness of the architectural implementation of the hardware memory model for granted.

7 Conclusions

We have presented an SMC approach for verifying concurrent programs under axiomatic weak memory models that keep track of dependencies between instructions, as is often the case with hardware memory models. Our experiments demonstrate that our tool performs much better than the few other tools supporting similar models, and yet suffers only moderate slowdown in comparison to the state-of-art tools supporting only porf-acyclic models.

In the future, we would like to find ways to further reduce this overhead as well as to extend the tool to support other low-level hardware features, such as cache maintenance instructions and mixed-sized memory accesses. Another item for future work is developing automated verification approaches for more advanced memory models, such as Promising [25] and Weakestmo [14], that allow “load buffering” outcomes without recording dependencies between program instructions.

Acknowledgments

We would like to thank Christopher Pulte, Azalea Raad, and the ASPLOS'20 reviewers for their valuable feedback.

References

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

[2] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2017. Source sets: A foundation for optimal dynamic partial order reduction. *J. ACM*, 64, 4, Article 25, (Sept. 2017), 25:1–25:49. doi: 10.1145/3073408.

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

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.*, 2, OOPSLA, Article 135, (Oct. 2018), 135:1–135:29. doi: 10.1145/3276505.

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

[6] Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. 2018. Constrained dynamic partial order reduction. In *CAV 2018*. Hana Chockler and Georg Weissenbacher, (Eds.) Springer International Publishing, Cham, 392–410. doi: 10.1007/978-3-319-96142-2_24.

[7] Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial orders for efficient bounded model checking of concurrent software. In *CAV 2013* (LNCS). Vol. 8044. Springer, Berlin, Heidelberg, 141–157. doi: 10.1007/978-3-642-39799-8_9.

[8] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In *ASPLOS 2018*. ACM, New York, NY, USA, 405–418. doi: 10.1145/3173162.3177156.

[9] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36, 2, Article 7, (July 2014), 7:1–7:74. doi: 10.1145/2627752.

[10] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. 2018. Optimal dynamic partial order reduction with observers. In *TACAS 2018* (LNCS). Vol. 10806. Springer, 229–248. doi: 10.1007/978-3-319-89963-3_14.

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

[12] James Bornholt and Emina Torlak. 2017. Synthesizing memory models from framework sketches and litmus tests. In *PLDI 2017*. ACM, New York, NY, USA, 467–481. doi: 10.1145/3062341.3062353.

[13] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI 2007*. ACM, New York, NY, USA, 12–21. doi: 10.1145/1250734.1250737.

[14] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.*, 3, POPL, Article 70, (Jan. 2019), 70:1–70:28. doi: 10.1145/3290383.

[15] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. Data-centric dynamic partial order reduction. *Proc. ACM Program. Lang.*, 2, POPL, Article 31, (Dec. 2017), 31:1–31:30. doi: 10.1145/3158119.

[16] Ken Harold. 2014. Choosing linux for medical devices. [Online; accessed 16-August-2019]. (2014). https://www.windriver.com/whitepapers/choosing-linux-for-medical-devices/White_Paper_Choosing_Linux_for_Medical_Devices.pdf.

[17] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *TACAS 2004* (LNCS). Vol. 2988. Springer, Berlin, Heidelberg, 168–176. doi: 10.1007/978-3-540-24730-2_15.

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

[19] Natalia Gavrilenko, Hernán Ponce-de-León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for weak memory models: Relation analysis for compact SMT encodings. In *CAV 2019*. Isil Dillig and Serdar Tasiran, (Eds.) Springer International Publishing, Cham, 355–365. doi: 10.1007/978-3-030-25540-4_19.

[20] Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *POPL 1997*. ACM, New York, NY, USA, 174–186. doi: 10.1145/263699.263717.

[21] Patrice Godefroid. 2005. Software model checking: The VeriSoft approach. *Form. Meth. Syst. Des.*, 26, 2, (Mar. 2005), 77–101. doi: 10.1007/s10703-005-1489-x.

[22] Patrice Godefroid, Robert S. Hanmer, and Lalita Jategaonkar Jagadeesan. 1998. Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using VeriSoft. In *ISSTA 1998*. ACM, New York, NY, USA, 124–133. doi: 10.1145/271771.271800.

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

[24] Shiyou Huang and Jeff Huang. 2016. Maximal causality reduction for TSO and PSO. In *OOPSLA 2016*. ACM, New York, NY, USA, 447–461. doi: 10.1145/2983990.2984025.

[25] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL 2017*. ACM, New York, NY, USA, 175–189. doi: 10.1145/3009837.3009850.

[26] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.*, 2, POPL, Article 17, (Dec. 2017), 17:1–17:32. doi: 10.1145/3158105.

[27] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model checking for weakly consistent libraries. In *PLDI 2019*. ACM, New York, NY, USA, 15 pages. doi: 10.1145/3314221.3314609.

[28] Michalis Kokologiannakis and Konstantinos Sagonas. 2019. Stateless model checking of the linux kernel's read–copy update (RCU). *Int. J. Soft. Tool. Tech. Transf.*, (Mar. 2019). doi: 10.1007/s10009-019-00514-6.

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

[30] Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28, 9, (Sept. 1979), 690–691. doi: 10.1109/TC.1979.1675439.

[31] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and reproducing heisenbugs in concurrent programs. In *OSDI 2008*. USENIX Association, 267–280.

[32] Brian Norris and Brian Demsky. 2013. CDSChecker: Checking concurrent data structures written with C/C++ atomics. In *OOPSLA 2013*. ACM, 131–150. doi: 10.1145/2509136.2509514.

[33] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *TPHOLs 2009*. Springer, 391–407. doi: 10.1007/978-3-642-03359-9_27.

[34] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3, POPL, Article 69, (Jan. 2019), 69:1–69:31. doi: 10.1145/3290382.

[35] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.*, 2, POPL, 19:1–19:29. doi: 10.1145/3158107.

[36] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: A simpler and faster operational concurrency model. In *PLDI 2019*. ACM, New York, NY, USA, 1–15. doi: 10.1145/3314221.3314624.

[37] 2009. rmem: Executable concurrency models for ARMv8, RISC-V, Power, and x86. [Online; accessed 24-August-2019]. (2009). https://github.com/rems-project/rmem.

[38] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. 2015. Unfolding-based partial order reduction. In *CONCUR 2015* (LIPIcs). Vol. 42. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 456–469. doi: 10.4230/LIPIcs.CONCUR.2015.456.

[39] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *PLDI 2011*. ACM, 175–186. doi: 10.1145/1993498.1993520.

[40] SPARC International Inc. 1994. *The SPARC architecture manual (version 9)*. Prentice-Hall.

[41] SV-COMP. 2019. Competition on software verification (SV-COMP). [Online; accessed 27-March-2019]. (2019). https://sv-comp.sosy-lab.org/2019/.

[42] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2017. Tricheck: Memory model verification at the trisection of software, hardware, and ISA. In *ASPLOS 2017*. ACM, New York, NY, USA, 119–133. doi: 10.1145/3037697.3037719.

[43] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual Volume I: User-level ISA*. https://content.riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf.

[44] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *POPL 2017*. ACM, 190–204. doi: 10.1145/3009837.3009838.

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

A Artifact Appendix

A.1 Abstract

We consider our paper's artifact to be the set of benchmarks we used in the paper, as well as the results we got by running particular versions of model checking tools (GenMC, HMC, Nidhugg, Herd, rmem, Dartagnan) on the benchmarks set. We do *not* consider the artifact of the paper to be HMC, as it will evolve over time, and the results obtained by running the same benchmarks may differ in the future.

We have also made HMC publicly available on GitHub as part of the GenMC tool (https://github.com/MPI-SWS/genmc). For any bugs, comments, or feedback regarding or HMC, please do not hesitate to contact us.

A.2 Artifact check-list (meta-information)

- **Algorithm:** HMC.
- **Program:** C benchmarks publicly available at GenMC's repository.
- **Run-time environment:** VirtualBox.
- **Output:** Console.
- **Experiments:** Scripts that fully reproduce the paper's results are provided.
- **How much disk space required (approximately)?:** After unpacking the VM image, approximately 10 GB.
- **How much time is needed to prepare workflow (approximately)?:** Everything is already set up.
- **How much time is needed to complete experiments (approximately)?:** <24 hours (see Section A.6).
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** See the pages of the tools for the respective licenses. For all code not explicitly covered by these licenses, GPLv2 applies.
- **Data licenses (if publicly available)?:** GPLv2.
- **Archived (provide DOI)?:** 10.5281/zenodo.3562082

A.3 Description

A.3.1 How delivered

The artifact is available on Zenodo (https://doi.org/10.5281/zenodo.3562082) and consists of a Virtual Machine (VM) containing binaries for all the model checking tools we used, along with all the benchmarks used in the submitted version of our paper, and HMC's source code. These hopefully suffice to validate the claims made in the paper.

A.3.2 Hardware dependencies

None in particular; allocating at least 2GB of RAM for the VM is recommended but not strictly required. Depending on the VirtualBox version there might be restrictions on the CPU (see VirtualBox's manual; https://download.virtualbox.org/virtualbox/6.0.14/UserManual.pdf).

A.3.3 Software dependencies

An operating system in which VirtualBox can be installed (see VirtualBox's manual; https://download.virtualbox.org/virtualbox/6.0.14/UserManual.pdf).

A.4 Installation

1. Download and install VirtualBox (https://www.virtualbox.org/wiki/Downloads), in case it is not already installed. We have tested the VM with VirtualBox 6.0.14 under Debian GNU/Linux.
2. Open VirtualBox, and import our VM by clicking "File" and then "Import Appliance".
3. After starting the VM, you can log in with the username "user" and password "hmc". Once logged in, shortcuts to the terminal and the file manager can be found under "Activities".

A.5 Experiment workflow

The results of the paper are reproduced using some bash scripts which print out some tables corresponding to the ones in our paper.

A.6 Evaluation and expected result

For the following sections we assume that the working directory is `~/asplos20-benchmarks`.

For all tables, we use a clock symbol to denote a timeout, and an X symbol to denote a failure of some sort. As in the paper, the timeout limit is set to 30 minutes. Note that, depending on the RAM the VM is allocated, some entries where a timeout is expected might be shown as failed instead (e.g., for Table 2). This is usually due to a stack overflow, and can happen for either Herd or rmem.

A.6.1 Reproducing Table 1 (∼ 5.3 hours)

To reproduce the results of Table 1, please issue the following command:

```
./get-table1.sh
```

A.6.2 Reproducing Table 2 (∼ 12.5 hours)

To reproduce the results of Table 2, please issue the following command:

```
./get-table2.sh
```

A.6.3 Reproducing Table 3 (∼ 1.5 hours)

Similarly, to reproduce the results from Table 3 issue:

```
./get-table3.sh
```

A.6.4 Reproducing Table 4 (∼ 1 hour)

To reproduce the results of Table 4 issue:

```
./get-table4.sh
```

A.6.5 Reproducing Table 5 (∼ 30 minutes)

To reproduce the results of Table 5 issue:

```
./get-table5.sh
```

A.6.6 Reproducing the overhead measurements

For this particular subsection, we assume that the working directory is `~/asplos20-benchmarks/plots`.

All the data we used for the scatter diagram and the overhead measurements can be found at the `scatter.dat` file. These are obtained from GenMC's standard test suite results, as well as from the benchmarks used in this paper.

To get a PDF file containing the scatter diagram please issue:

```
latexmk -pdf main.tex && evince main.pdf
```

To get the geometric means of Figure 2, please issue the following commands:

```
./geo-mean.awk scatter.dat 3 7 0.1
./geo-mean.awk scatter.dat 7 5 0.1
./geo-mean.awk scatter.dat 3 5 0.1
```

By invoking `max-overhead.awk` instead of `geo-mean.awk` you can get the maximum overheads instead.

Both these scripts as arguments the name of the data file, the column over which we are normalizing, the overhead column, and the threshold below which we do not consider entries.

A.7 Experiment customization

A.7.1 Running HMC/GenMC

A generic invocation of GenMC looks like the following:

```
genmc [OPTIONS] -- [CFLAGS] <file>
```

Where `CFLAGS` are options that will be passed directly to the C compiler, and `OPTIONS` include several options that can be passed to GenMC. Among these options, the most useful ones are probably the `-unroll=N` switch, which unrolls a loop `N` times, and the `-wb` and `-mo` options, that enable the WB and the MO variant of GenMC, respectively (default is WB). Lastly, `file` should be a C file that uses `pthreads` for concurrency.

To use HMC, please invoke GenMC with the `-imm` option. More information regarding the usage of the tool can be found at the tool's manual (https://github.com/MPI-SWS/genmc/tree/master/doc).

A.7.2 Available benchmarks

The benchmarks we used for the tables of our paper are located in the directory `~/asplos20-benchmarks/benchmarks`. Apart from the benchmarks located in the folder above, many more benchmarks can be found at GenMC's repository (https://github.com/MPI-SWS/genmc/tree/master/tests).

In the above repository and the relevant sub-directories, there is a separate folder for each benchmark, that contains the "core" of the test case, as well as the expected results for the test case, some arguments necessary for the test case to run, etc. In order to actually run a test case, we can run the tool with one of the test case variants, which are located in a folder named 'variants', in turn located within the respective test case's folder.

For example, assuming that GenMC's repository has been cloned at REPO, to run a simple Store Buffering (SB) test case with HMC, please issue:

```
genmc -imm REPO/tests/correct/synthetic/SB/variants/sb0.c
```