Model Checking the XMM Memory Model

Matteo Meluzzi

Thesis committee:

Chair:	Prof. dr. Arie van Deursen	TU Delft
Supervisor:	Dr. Soham Chakraborty	TU Delft

submitted in partial fulfillment of the requirements for the degree of
Master of Science
in
Computer Science

Abstract

XMM is a newly designed multi-execution memory model that solves the out-of-thin-air executions problem, enables the most efficient compilation to all hardware platforms, and allows common compiler optimizations. Promising and Weakestmo are similar multi-execution models proposed recently. However, due to their complex semantics, they do not have an accompanying model checker with a proof of soundness. XMM is significantly less complex, which paved the way for implementing an XMM model checker.

In our work, we present our design of a model checker algorithm for the XMM memory model called GenMC-XMM. Due to practical limitations of algorithmic time complexity, we could not design a complete model checker. However, we provide a proof of soundness. In other words, we cannot guarantee that our tool will find all possible executions of a program, but we can guarantee that the ones it finds are all XMM consistent. To our knowledge, GenMC-XMM is the first multi-execution model checker proven to be sound. We evaluated our tool against the state-of-the-art model checking tools for RC11, IMM, and Weakestmo2. We determined that GenMC-XMM explores the same or more executions than every other tool in all our tests. GenMC-XMM is only marginally slower than the other tools in real-world lock-free data-structure benchmarks. In synthetic tests designed to have thousands of data races, GenMC-XMM does not scale as well as the other tools as the number of races increases.

GenMC-XMM is the next milestone in the automatic verification of multi-execution memory models. Based on our work on GenMC-XMM, other researchers may improve its performance and design a sound and complete algorithm.

Preface

I am deeply grateful to Soham and Evgenii for guiding me while I was working on this project. I could not have done it without their invaluable insights, suggestions, and feedback.

To Arie who agreed to chair my thesis committee and took the time to listen to my midterm presentation, green light review, and thesis defence.

To Blanca and my uncle Alessandro for reading my thesis and giving me feedback.

To my friends Bobe, Alex, Chelsea, and Bogdan, whom I met daily to work together on our theses. They made it a very enjoyable routine.

To my family for their unconditional support, constant encouragement, and belief in me.

Matteo Meluzzi
Delft, June 2024

Contents

List of Figures

List of Tables

Chapter 1

Introduction

How can software engineers design lock-free concurrent data structures with compiler optimizations while ensuring consistent program behavior, irrespective of the compiler used and target architecture? Verifying the code with a relaxed memory model checker is one solution to this challenging engineering problem.

Memory models are a formal definition of the concurrency behavior of a program. Modern programming languages often have a built-in memory model to ensure consistent compilation across hardware architectures. The adjective *relaxed* refers to a memory model that allows memory operations to not strictly follow the order they are written in the source code. On the one hand, this enables the compiler and the hardware platform to perform optimizations, but on the other, it can confuse the programmer with odd program behaviors. The role of the memory model is to establish a common ground between the programmer, compiler, and hardware architectures on what program outcomes are possible. Examples of standard relaxed memory models are TSO [32] (Total Store Order), Arm [37], and RC11 [24]. The model that only allows interleavings of memory operations without reorderings is SC (Sequential Consistency), and it is the most intuitive to the programmer but allows a minimum amount of optimizations.

There exist two classes of memory models: per-execution models and multi-execution models. The per-execution class is closer to SC and, therefore, simpler. The multi-execution class deviates significantly from SC, even allowing cyclic execution graphs. This increases their complexity but also enables more program optimizations. The complexity is reflected in their accompanying model checker tools, which have not been proven sound or complete as of our writing [28, 30, 36].

In the per-execution approach, multiple programs can share an execution graph. Depending on the consistency constraints, an execution may be permitted or prohibited. Prohibiting the execution could prevent a valid program execution, which affects possible program optimizations. On the other hand, allowing the execution might lead to an out-of-thin-air execution in another program. An example of this trade-off are the C11 and RC11 models. C11 allows out-of-thin-air executions in certain cases. RC11 forbids all cyclic executions and prevents some optimizations like read-write reordering, resulting in sub-optimal compilation to Arm, Power, and RISCV due to extra fences needed.

In the multi-execution approach, each program is identified by all of its possible executions. Typically, multi-execution models allow cyclic execution graphs, whereas per-execution models do not. This allows multi-execution models to surpass the limitations of per-execution models but adds complexity because multiple executions are examined together to derive another consistent execution.

In the past decade, there have been several attempts at defining a multi-execution model that could be easily verified automatically by a model checker. *Promising Semantics* [19] has a verification tool for part of the model [36]. *Weakestmo* [9] has a model checking algorithm, called *WMC* [28] for a slightly strengthened version of the model. Neither of them has been proven to be sound or complete. Despite the effort to manage their complexity, these models and model checkers are difficult to reason about.

We introduce a new, simpler model checker algorithm for the XMM memory model [34] called GenMC-XMM. Our model checker is simpler than the current state-of-the-art multi-execution models because the XMM model lends itself to be checked in a single-execution style while at the same time allowing for weak behaviors that permit the compiler to perform common optimizations. Due to algorithmic time complexity limitations, we could not design GenMC-XMM to be complete. However, we prove that GenMC-XMM is sound. In other words, GenMC-XMM is not guaranteed to output all XMM consistent executions, but all the outputted ones are guaranteed to be XMM consistent. In our evaluation of GenMC-XMM, we found that its performance is on par with the current state-of-the-art model checkers in tests without data races. In tests with data races, GenMC-XMM's performance is not as good as that of tools limited to acyclic executions.

In this work, we discuss the state-of-the-art GenMC [21] model checker tool upon which GenMC-XMM is based and the WMC [28] model checker due to its many similarities with GenMC-XMM. We give an overview of the XMM model. We get into the details of the design of GenMC-XMM. Lastly, we compare its performance and behavior with other state-of-the-art memory model checker tools.

For all the examples in this paper, we assume that variables are initialized to 0 and that memory accesses are *relaxed* (as defined in the C++ memory ordering: "there are no synchronization or ordering constraints imposed on other reads or writes, only this operation's atomicity is guaranteed").

Chapter 2

Background

In this chapter, we discuss background knowledge that is useful for understanding the remainder of this paper. We discuss the sequential consistency memory model, the C++ memory model, execution graphs, and load buffering races (LB races).

2.1 Sequential Consistency

The concept of sequential consistency was first introduced by Leslie Lamport in 1979 [25]: "Consider a computer composed of several such [sequential] processors accessing a common memory. The customary approach to designing and proving the correctness of multiprocess algorithms for such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. A multiprocessor satisfying this condition will be called sequentially consistent."

A possible mental model that can be used to imagine a sequentially consistent computer is a machine with multiple threads, one shared memory that serves one thread at a time, and no caches. In this model, the two following properties hold:

1. There are no local reorderings, meaning that the instructions of a single thread are executed exactly in the order they are written in the program.
2. Every write becomes immediately available to all other threads.

Sequential consistency is the most intuitive model for programmers since simply interleaving the threads without reordering operations can explain a program outcome.

Consider the following Store Buffer litmus test:

$$\begin{array}{l|l} X := 1 & Y := 1 \\ r1 := Y & r2 := X \end{array}$$

There are only six possible executions of this program on a sequentially consistent machine:

$X := 1$			$X := 1$			$X := 1$	
$r1 := Y$				$Y := 1$			$Y := 1$
	$Y := 1$		$r1 := Y$				$r2 := X$
	$r2 := X$			$r2 := X$		$r1 := Y$	
	$Y := 1$			$Y := 1$			$Y := 1$
$X := 1$				$r2 := X$		$X := 1$	
$r1 := Y$			$X := 1$				$r2 := X$
	$r2 := X$		$r1 := Y$			$r1 := Y$	

Figure 2.1: Sequentially consistent thread interleavings of Store Buffer test

From Figure 2.1, we can determine that there is no sequentially consistent execution where $r1 = 0, r2 = 0$. However, this outcome could occur if this program were executed on a modern multiprocessor computer without special synchronization instructions. All modern multiprocessor computers have registers and caches, and they can execute some instructions out-of-order. This is very important for performance, but as a consequence, no modern computer operates under the SC model. However, it is possible to compile this program to appear as though it is executing in a sequentially consistent manner because all architectures provide special synchronization instructions. These instructions allow us to write concurrent programs. Still, they are often quite expensive to use in terms of CPU cycles and cache misses, so their use should be kept to a minimum while maintaining program correctness.

2.2 The C++ Memory Model

The C++ language received its first memory model in 2011 with the release of the C++11 standard. The C11 model defines atomic variables that, by default, behave according to the Sequential Consistency model. Other memory order options are *release/acquire* and *relaxed* semantics.

An *acquire load* is guaranteed that no subsequent reads or writes can be reordered before it and that if a *same-location release store* was executed in another thread, all writes in that thread become visible to the acquire load's thread. Similarly, a *release store* is guaranteed that no previous reads or writes can be reordered after it and that all writes in its thread will be visible to other threads that perform a *same-location acquire load*.

A *relaxed* memory access does not impose any synchronization or ordering constraints. Only its atomicity is guaranteed. Relaxed memory accesses proved to be not well-defined in the C11 memory model because it allows out-of-thin-air (OOTA) values in some cases, and certain common compiler optimizations proved to be incorrect [41]. Out-of-thin-air values are particularly harmful to program reasoning because they create self-validating cycles that allow any value to be read "out-of-thin-air". In Figure 2.2, no rule in the C++11 model prevents $r1$ and $r2$ from having any value. The consequence of "out-of-thin-air" values is that the C++ model can not be effectively model-checked by an algorithm without some form of patching for this problem. Notably, the RC11 (Repaired C11) [24] model is commonly used and solves this issue by forbidding all cyclic executions. Consequently, some optimizations, such as load-store reordering, are not allowed under RC11. In figure Figure 2.3, we show one such example.

$$r_1 = X; \quad \| \quad r_2 = Y;$$
$$Y = r_1; \quad \| \quad X = r_2;$$

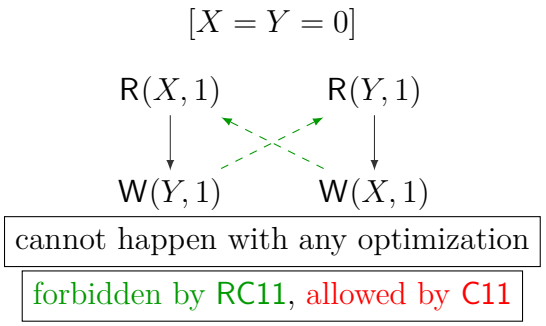


Figure 2.2: Program that causes an OOTA execution to be allowed by C11, but forbidden by RC11.

$$r_1 = X; \quad \| \quad r_2 = Y;$$
$$Y = 1; \quad \| \quad X = r_2;$$

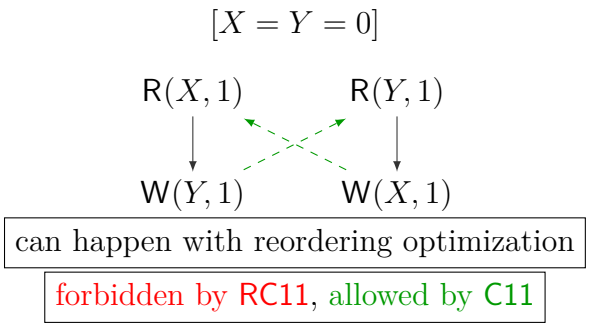


Figure 2.3: Program that causes a valid cyclic execution to be allowed by C11, but forbidden by RC11.

2.3 Execution Graphs

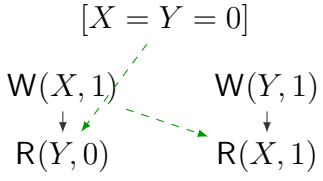
The possible outcomes of a program can be represented as a set of *execution graphs*. Each graph has events (nodes) of type read (R) or write (W) and edges of type po and rf:

- po (program order) edges relate events in the same thread based on the order in which they appear in the program's source code. They are represented as solid black arrows.
- rf (reads-from) edges relate a read to the write from which it reads. They are represented as dashed green arrows.

For example, the Store Buffer test we showed in section 2.1 had six possible program interleavings under SC. Each of the interleavings can be represented as an execution graph. We show two examples below:

$$X := 1 \quad \| \quad Y := 1$$
$$r1 := Y \quad \| \quad r2 := X$$

$$X := 1$$
$$r1 := Y$$
$$Y := 1$$
$$r2 := X$$



$$
\begin{array}{ll|c}
X := 1 & & [X = Y = 0] \\
 & Y := 1 & \mathsf{W}(X,1) \quad \mathsf{W}(Y,1) \\
r1 := Y & & \downarrow \quad \downarrow \\
 & r2 := X & \mathsf{R}(Y,1) \quad \mathsf{R}(X,1)
\end{array}
$$

Additional useful relations defined on execution graphs are:

- mo (memory order or equivalently co coherence order) orders two same-location writes.
- sw (synchronizes-with) a release event a *synchronizes with* an acquire event b, whenever b (or, if b is a fence, some po-prior read) reads from the release sequence of a (or, if a is a fence, of some po-later write).
- hb (happens-before) an event a *happens-before* event b if there is a path between a and b consisting of po and sw edges.
- fr (from-read) relates a read event $\mathsf{R}(X,i)$ to a same-location write event $\mathsf{W}_{\text{after}}(X,j)$ if $\mathsf{W}(X,i) \xrightarrow{\mathsf{rf}} \mathsf{R}(X,i) \wedge \mathsf{W}(X,i) \xrightarrow{\mathsf{mo}} \mathsf{W}_{\text{after}}(X,j)$.
- rmw (read-modify-write) relates a read event to a po-immediate same-location write event when they correspond to a read-modify-write atomic operation.

The following operations can be used on a relation $R \subseteq E \times E$:

- R^{-1} is the inverse relation of R. $R^{-1} = \{(y,x) \mid (x,y) \in R\}$
- $R^?$ is the reflexive closure of R: $R^? = R \cup \{(x,x) \mid x \in E\}$
- $R_1; R_2$ is the sequential composition operation: $R_1; R_2 = \{(x,y) \mid \exists z \text{ s.t. } (x,z) \in R_1 \wedge (z,y) \in R_2\}$.
- $R_1 \cup R_2$ is the union operation: $R_1 \cup R_2 = \{(x,y) \mid (x,y) \in R_1 \vee (x,y) \in R_2\}$.
- The sequential composition operator (;) takes precedence over union ($\cup$): $R_1 \cup R_2; R_3 = R_1 \cup (R_2; R_3)$
- R is irreflexive if it does not relate any element to itself: $\nexists x \text{ s.t. } (x,x) \in R$.

2.4 Load Buffering Data Races

For the remainder of this paper, we will use the concepts of load buffering data race (LB race) and load buffering race freedom (LB race freedom). We begin by defining the related concept of data race.

Definition 2.4.1 (Data Race)**.** Two concurrent events are in *race* if they access the same location, and at least one of them is a write event. Let G.Race be the set of all racy events of execution G:

$$\mathsf{G.Race} \triangleq (\mathsf{G.E} \times \mathsf{G.E}) \setminus \mathsf{G.hb}^{=})|_{\mathsf{loc}} \cap \mathsf{one}(\mathcal{W})$$

where $\mathsf{one}(A)$ is the relation where at least one of its components belongs to the set A:

$$\mathsf{one}(A)(x,y) \triangleq (x \in A \vee y \in A).$$

Let $\mathsf{G.Race}(o) \subseteq \mathsf{G.Race}$ be the set of races where at least one of the involved events has a weaker or equal memory order than o. We define $\mathsf{G.Race}(o)$ as follows:

$$\mathsf{G.Race}(o) \triangleq \mathsf{G.Race} \cap \mathsf{one}(\mathsf{G.E}_{\sqsubseteq o})$$

Definition 2.4.2 (Load buffering race)**.** A pair of events r and w form a *load buffering race (LB race)* in an execution graph G if r is a read, w is a concurrent write to the same location, and there is a porf path from r to w

$$\mathsf{LBRace} \triangleq \mathsf{Race}(\mathsf{Rlx}) \cap ([\mathsf{R}]; \mathsf{porf}; [\mathsf{W}])$$

Definition 2.4.3 (Load buffering race-free program)**.** A program P is *LB-race-free* under a memory model M if no consistent execution graph of P under M has a load buffering race.

Definition 2.4.4 (Load buffering race freedom)**.** A memory model M provides the *load buffering race freedom* guarantee with respect to a stronger memory model M' (written as $\mathsf{LBRF}(M')$) if, for any LB-race-free program P under M', its consistent executions under M are exactly the same as under M'.

Chapter 3

GenMC-XMM's parent: GenMC

Suppose that we would like to verify that a concurrent program always satisfies the assertions contained in it or whether it is data-race-free. An effective way of solving this problem is to use a *stateless model checker* like GenMC [20], which enumerates all possible executions according to a memory model and checks each one. One issue that model checkers face is that the set of valid executions depends on the targeted memory model. For example, an execution might be valid under C11 [6] but not under SC (sequential consistency). GenMC tackles this issue by being parametric on the memory model choice. It supports SC, RC11 [24], and IMM [33]. GenMC is designed to be extended to work with any model under the assumption of porf-acyclicity.

3.1 Overview

GenMC takes as input a C++ program, interprets it with llvm, and constructs execution graphs with events, po edges, rf edges, and mo ordering. GenMC incrementally records which reads can be modified to read from a different write as the graphs get built. Once an execution graph is constructed, GenMC picks one of these reads, changes its rf edge, and re-executes the program. The authors of GenMC have proved that their algorithm is sound: no false positives, complete: explores all executions, and optimal: no execution is visited twice [20].

Due to the high number of executions a program can generate, maintaining an in-memory graph for each execution is costly. GenMC approaches this issue by memorizing only which read and write should be revisited instead of immediately generating a new graph. To this end, GenMC maintains a work queue for each graph. A work queue is a list of pairs of reads and writes that should be revisited.

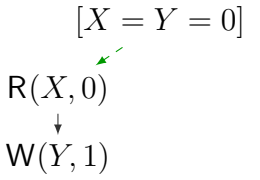
When a graph G is fully constructed, GenMC picks a revisit $\langle R, W \rangle$ from the work queue of G. G is restricted to only events added to the graph after R, resulting in graph G'. Finally, G' is re-executed, and the process is repeated. If a revisit was added to the work queue due to a write added to G, that revisit is a *backward revisit*. If it was added because of a new read, then it is a *forward revisit*. These two classes of revisits differ in that backward revisits can lead to duplicated execution and should not be removed from the work queue but instead marked as "explored". Forward revisits, on the other hand, do not lead to duplication and can safely be removed.

3.2 Step-by-Step Example

Let us run the $\text{GenMC}_{\text{RC11}}$ algorithm on the Load Buffering litmus test shown below.

$$\begin{array}{l|\!|l} a := X & b := Y \\ Y := 1 & X := b \end{array}$$

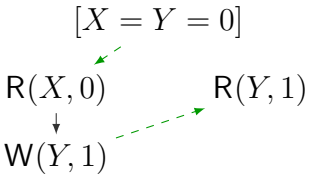
We begin with a graph containing only the initialization writes $\mathsf{W}(X,0)$ and $\mathsf{W}(Y,0)$ represented by $[X = Y = 0]$. Suppose we execute threads from left to right, starting from thread one.



Work queue:

$\emptyset$

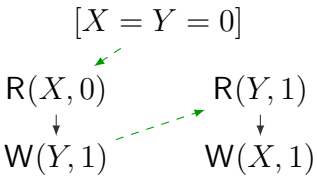
Now thread one has completed, so we schedule thread two and run $b := Y$



Work queue:

$\langle \mathsf{R}(Y,1), \mathsf{W}(Y,0) \rangle$

The newly added $\mathsf{R}(Y,1)$ can either *read-from* $\mathsf{W}(Y,1)$ or from the initialization write $\mathsf{W}(Y,0)$. The former case is represented directly in the graph. The latter is recorded in the work queue. Since the event we just added is a read, it is a forward revisit.

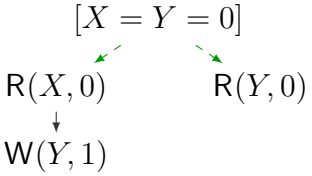


Work queue:

$\langle \mathsf{R}(Y,1), \mathsf{W}(Y,0) \rangle$

We continue execution and reach statement $X := b$, which adds $\mathsf{W}(X,1)$ to the graph. We do not add a backward revisit $\langle \mathsf{R}(X,0), \mathsf{W}(X,1) \rangle$ to the work queue because $\mathsf{W}(X,1)$ is in the porf-suffix of $\mathsf{R}(X,0)$ which would result in a cycle.

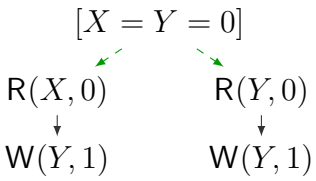
We reached the end of the execution, so we pop an item from the work queue. We pop $\langle R(Y,1), W(Y,0) \rangle$. The next step is restricting the graph to only events added before the read event. The rf edge of $R(Y,1)$ is changed to read-from the initialization write $W(Y,0)$.



Work queue:

$\emptyset$

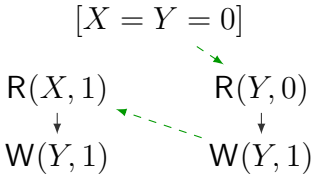
We reach statement $X := b$, which results in $\mathsf{W}(X,1)$ to be added to the graph. $\mathsf{R}(X,0)$ can read from it without creating a cycle, so we record the backward revisit $\langle \mathsf{R}(X,0), \mathsf{W}(X,1) \rangle$ in the work queue.



Work queue:

$\langle \mathsf{R}(X, 0), \mathsf{W}(X, 1) \rangle$

We have found another complete execution graph. Once again, we pop an item from the work queue, restrict the graph, and re-execute. Since $\langle R(X, 0), W(X, 1) \rangle$ is a backward revisit, we do not remove it. Instead, we mark it as "explored". The result is our third and last complete execution graph:



Work queue:

$\langle R(X, 0), W(X, 1) \rangle$ explored

3.3 Assumptions

According to [20], GenMC makes the following four assumptions on the underlying memory model:

1. porf **acyclicity**: this requirement is satisfied by models like SC, TSO, PSO, and RC11. The C11 model does not meet it and sometimes allows problematic out-of-thin-air (OOTA) values to be generated from porf cycles. Other weak models such as ARM, Power, Weakestmo, Promising, and XMM do not satisfy the condition but avoid OOTA values.

2. **prefix-closedness**: *There exists a partial order R that includes reads-from and (preserved) program order such that, if a graph is consistent, so is every R-prefix of it.* Thanks to this property, GenMC can check consistency every time an event is added to a graph, which is a more efficient approach compared to checking only once a graph is complete.

3. **extensibility**: *Given a consistent execution G, a* po*-maximal event can always be added to G to yield a consistent execution (with an appropriate* rf *edge when applicable).* Thanks to this property, GenMC can be certain that discarding an inconsistent execution will not lead to missed exploration options.

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

Chapter 4

GenMC-XMM's sibling: WMC

WMC [28] is a model checker designed for the Weakestmo [9] multi-execution memory model. Multi-execution style models differ from per-execution in that they consider multiple executions together to evaluate program outcomes. This has the benefit of allowing weak program behaviors that enable the compiler to perform optimizations. Still, their formal semantics are complex, and therefore also designing a model checker for them. Similarly to Promising Semantics [19], WMC generates weak out-of-order behaviors by letting threads promise to write a value speculatively, as long as they can justify the promise with an execution that proves that it is possible to execute that write.

To facilitate the design of a model checker, the authors of the WMC paper introduced two properties to Weakestmo: *load buffering race freedom* (LBRF_{RC11}) and *certification locality* (CL), and called this strengthened model Weakestmo2. The first property states that there can only be cyclic executions in the presence of Load Buffering Data Races (LB races). Since Weakestmo2 satisfies LBRF_{RC11}, Weakestmo2 is equivalent to RC11 for LB-race-free programs. The second property states that once a thread depends on an external (from another thread) write to justify a promise, it should not be able to ignore that write in favor of a new one. This property rules out certain OOTA behaviours. The authors show that Weakestmo2 preserves the efficient compilation mappings [29] and the soundness of local program transformations [9] of Weakestmo.

The Weakestmo model uses event structures [43] to represent multiple program outcomes in a single graph: they can contain several execution branches that can be used to analyze the out-of-order executions. An execution graph G is *Weakestmo-Consistent* if there exists a *Weakestmo-Consistent* event structure S such that G can be extracted from S. In Weakestmo2, a graph is consistent if it is *Weakestmo-Consistent* and satisfies *certification locality*.

4.1 Algorithm Outline

WMC is based on the GenMC model checker (Chapter 3), adapted to the Weakestmo2 memory model as follows:

1. When calculating the set of revisitable reads, GenMC excludes those porf-before the revisiting write. WMC includes them if there is an LB race between the read and the revisiting write.
2. When revisiting a write w and a porf-earlier read r, WMC restricts the graph to remove the porf-prefix of w that is po-after r. The writes that are po-after r and are read externally

are kept in a *promise set* that consists of pairs of promised writes and the execution graph as it was when the promise was issued.

3. WMC enters a certification phase when a new event is added to the graph, and the promise set is not empty. In this phase, all non-local explorations are postponed until all promises are fulfilled.
4. At the end of the certification phase, WMC calculates a new promise set and the next set of revisits.

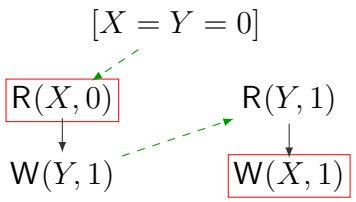
Unlike GenMC, WMC has to deal with duplicate executions being explored due to the cycles created in the execution graphs. WMC gives the option to either explore them again or memorize which graphs have already been explored in a hash map and discard them.

4.2 Step-by-Step Example

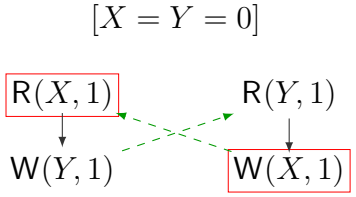
Let us run WMC on the Load Buffering litmus test:

$$\begin{array}{l|\!|l} a := X & b := Y \\ Y := 1 & X := b \end{array}$$

This is the first execution produced by WMC. It is acyclic, but it contains a Load Buffering race between $\mathsf{R}(X, 0)$ and $\mathsf{W}(X, 1)$.



Thread 1 can *promise* $\mathsf{W}(Y, 1)$. This allows $\mathsf{R}(Y, 1)$ to read from it when deriving the next graph.



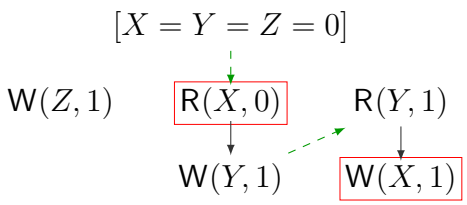
The last step is *promise certification*, which ensures that thread one can fulfill its promise of writing 1 to Y. Since in this example the promise is fulfilled, this execution is considered consistent.

4.3 Certification Locality

Certification Locality (CL) is a strengthening condition in the Weakestmo2 model, which forbids certain out-of-thin-air behaviors. The CL constraint ensures that read events created during promise certification can only read-from writes in the porf-prefix of the promise.

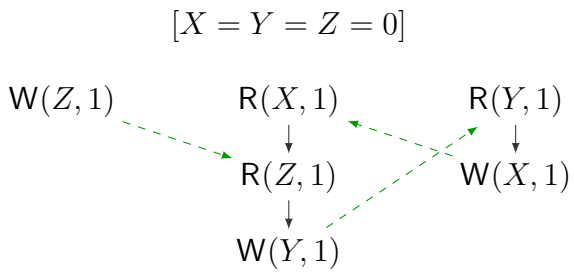
Take, for example, the following program and corresponding acyclic execution:

$$
Z := 1 \;\Big\|\; \begin{array}{l} \textbf{if}\,(X == 1)\,\{ \\ \quad Y := Z \\ \} \\ \textbf{else}\{ \\ \quad Y := 1 \\ \} \end{array} \;\Big\|\; \begin{array}{l} \textbf{if}\,(Y == 1)\,\{ \\ \quad X := 1 \\ \} \end{array}
$$



Assume that WMC did not enforce the CL property. If thread 2 promises $\mathsf{W}(Y, 1)$, this promise could be certified by taking the else branch and reaching $Y := 1$, or through the if branch when $Z == 1$, and reaching $Y := Z$.

However, in WMC, $\mathsf{W}(Z, 1)$ cannot be used for certifying the promised $\mathsf{W}(Y, 1)$ because $\mathsf{W}(Z, 1)$ is not in the porf-prefix of $\mathsf{W}(Y, 1)$ in the graph above. The following execution is, therefore, forbidden by WMC:



4.4 Results

The authors of WMC evaluated their algorithm and obtained the following (among other) results.

1. How often do LB races appear in practice? What overhead is there for detecting them?

   Realistic implementations of lock-free data structures rarely contain LB races. Out of 29 evaluated realistic data structures, only 2 had LB races. For LB race-free programs, the overhead of looking for the races in WMC was 25% more compared to GenMC.

2. How well does WMC perform against other tools that handle similar memory models?

   The authors found that WMC's performance is comparable to GenMC, CDSChecker [30], HMC [22], and Nidhugg [3] for some synthetic and data-structure benchmarks. The main overheads of WMC compared to GenMC are the search for LB races and the fact that Weakestmo2 allows a greater number of executions than RC11.

3. How does WMC scale in synthetic benchmarks containing many load buffering races?

   In a purposely crafted synthetic benchmark with many LB races, called *LBn-pairs(14)*, WMC explores as much as 61 741 duplicate executions out of 16 384 consistent ones. In this test, 14 threads are divided into seven pairs. Each pair reads and writes two variables assigned to it, creating an LB race. Since each pair has 3 RC11 executions and one cyclic execution, there are 4^n executions where n is the number of pairs.

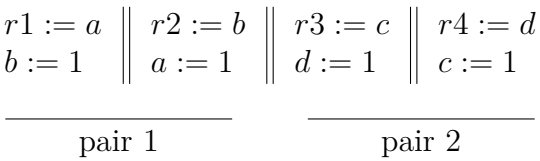


Figure 4.1: Example of LBn-pairs(4) test. Four threads are divided into two pairs. Each pair reads and writes two variables assigned to it.

Chapter 5

The XMM Memory Model

In this chapter, we discuss the XMM memory model [34] that GenMC-XMM is designed to model check. XMM is an axiomatic model with two operational steps that define how execution graphs can be built. The first step is *Execution*, which defines how RC11 acyclic graphs are constructed. To derive a cyclic graph from an RC11 graph, we arbitrarily choose a set of events to *commit* and perform the second step: *Re-Execution*. If an event is committed, it will be in the derived graph.

The operational semantics of graph construction use configurations and execution graphs defined as follows:

Definition 5.0.1 (Configuration)**.** A *configuration* is a tuple $\langle \mathsf{G}, \mathsf{G}', \mathcal{C} \rangle$ where G and G' are execution graphs, and $\mathcal{C} \subseteq \mathsf{G}'.\mathsf{E}$ is a set of *committed* events.

Definition 5.0.2 (Execution Graph)**.** An execution graph $\mathsf{G} \triangleq \langle \mathsf{E}, \mathsf{lab}, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{rmw} \rangle$ is a tuple, where E is a set of events, $\mathsf{lab} : \mathsf{E} \to \mathsf{LAB}$ is a labelling function, $\mathsf{po} \subseteq \mathsf{E} \times \mathsf{E}$ is the *program order*, $\mathsf{rf} \subseteq \mathsf{W} \times \mathsf{R}$ is the *reads-from* relation, $\mathsf{mo} \subseteq \mathsf{W} \times \mathsf{W}$ is the *modification order*, and $\mathsf{rmw} \subseteq \mathsf{R} \times \mathsf{W}$ is the *read-modify-write* relation. We say that the graph is *well-formed* if the following conditions are met:

$$
\begin{aligned}
&\mathit{WellFormed}(\mathsf{G}) \triangleq \\
&\quad \forall 0 \neq t \in \mathsf{TID}.\ \mathsf{G}.\mathsf{po}|_t \text{ is a total order} \\
&\quad \forall \langle e_1, e_2 \rangle \in \mathsf{po}.\ \mathsf{tid}(e_1) = 0 \wedge \mathsf{tid}(e_2) \neq 0 \vee \mathsf{tid}(e_1) = \mathsf{tid}(e_2) \\
&\quad \forall \langle w, r \rangle \in \mathsf{rf}.\ \mathsf{loc}(w) = \mathsf{loc}(r) \wedge \mathsf{val}(w) = \mathsf{val}(r) \\
&\quad \forall \langle w_1, r \rangle, \langle w_2, r \rangle \in \mathsf{rf}.\ w_1 = w_2 \\
&\quad \forall x \in \mathsf{loc}.\ \mathsf{G}.\mathsf{mo}|_x \text{ is total} \\
&\quad \forall \langle w_1, w_2 \rangle \in \mathsf{mo}(w_1, w_1).\ \mathsf{loc}(w_1) = \mathsf{loc}(w_2) \\
&\quad \mathsf{rmw} \subseteq \mathsf{imm}(\mathsf{po}) \quad \text{and} \quad \forall \langle r, w \rangle \in \mathsf{rmw}(r, w).\ \mathsf{loc}(a) = \mathsf{loc}(b)
\end{aligned}
$$

The formal rules of the XMM semantics are the following:

$$\frac{\mathsf{G} \xrightarrow{\langle e,\ell\rangle} \mathsf{G}' \qquad \mathit{RfComplete}(\mathsf{G}') \qquad \mathit{Consistent}(\mathsf{M},\mathsf{G}')}{\mathsf{G} \xrightarrow[\texttt{exec}]{\langle e,\ell\rangle} \mathsf{G}'}\ (\text{Execute})$$

$$\frac{\begin{array}{c}\mathsf{G}'.\mathsf{E} = \mathsf{G}.\mathsf{E} \uplus \{e\} \\ \mathsf{G}'.\mathsf{lab} = \mathsf{G}.\mathsf{lab}[e \mapsto \ell] \\ \mathsf{G}'.\mathsf{po} = \mathsf{G}.\mathsf{po} \cup \Delta_{\mathsf{po}}(\mathsf{G}, e) \\ \mathsf{G}'.\mathsf{rf} = \mathsf{G}.\mathsf{rf} \cup \Delta^{\mathsf{R}}_{\mathsf{rf}}(\mathsf{G}, w, e) \cup \Delta^{\mathsf{W}}_{\mathsf{rf}}(\mathsf{G}, R, e) \\ \mathsf{G}'.\mathsf{mo} = \mathsf{G}.\mathsf{mo} \cup \Delta_{\mathsf{mo}}(\mathsf{G}, W_1, W_2, e) \\ \mathsf{G}'.\mathsf{rmw} = \mathsf{G}.\mathsf{rmw} \cup \Delta_{\mathsf{rmw}}(\mathsf{G}, r, e)\end{array}}{\mathsf{G} \xrightarrow{\langle e,\ell\rangle} \mathsf{G}'}\ (\text{Add Event})$$

$$\frac{\begin{array}{cc}\mathcal{D} \subseteq \mathsf{f} \uparrow \mathcal{C} & \mathcal{C} \subseteq \mathsf{G}'.\mathsf{E} \\ \mathit{WellFormed}(\mathsf{G}|_{\mathcal{D}}, \mathsf{G}', \mathcal{C}) & \mathit{EmbeddedSubGraph}(\mathsf{G}, \mathsf{G}', \mathcal{C}, \mathsf{f}) \\ \langle \mathsf{G}', \mathcal{C}\rangle \vdash \mathsf{G}|_{\mathcal{D}} \Rightarrow^* \mathsf{G}' & \mathit{StableUncommitedReads}(\mathsf{G}', \mathcal{C})\end{array}}{\mathsf{G} \xrightarrow[\texttt{re-exec}]{\langle \mathsf{f},\mathcal{C}\rangle} \mathsf{G}'}\ (\text{Re-Execute})$$

$$\frac{\mathsf{G}_i \xrightarrow{\langle e,\ell\rangle} \mathsf{G}_{i+1} \qquad \mathit{WellFormed}(\mathsf{G}_{i+1}, \mathsf{G}', \mathcal{C})}{\langle \mathsf{G}', \mathcal{C}\rangle \vdash \mathsf{G}_i \xrightarrow{\langle e,\ell\rangle} \mathsf{G}_{i+1}}\ (\text{Guided Step})$$

Figure 5.1: Rules of XMM execution graph construction

Definition 5.0.3 (RfComplete)**.** Execution graph G is *reads-from complete*, denoted as $\mathit{RfComplete}(\mathsf{G})$, if each read event in the graph reads from some write event belonging to the same graph:

$$\mathsf{G}.\mathsf{R} \subseteq \mathsf{codom}(\mathsf{G}.\mathsf{rf}).$$

Definition 5.0.4 (Consistent)**.** The execution graph G is C11 consistent if the following conditions are met:

(Coherence) $\mathsf{G}.\mathsf{hb};\mathsf{G}.\mathsf{eco}^?$ is irreflexive;

(Atomicity) $\mathsf{G}.\mathsf{rmw} \cap (\mathsf{G}.\mathsf{fr};\mathsf{G}.\mathsf{mo}) = \emptyset$; and

(SC) $\mathsf{G}.\mathsf{psc}$ is acyclic.

The eco relation is defined as follows:

$$\mathsf{eco} = \mathsf{rf} \cup \mathsf{mo};\mathsf{rf}^? \cup \mathsf{fr};\mathsf{rf}^?$$

Definition 5.0.5 (WellFormed Configuration)**.** A configuration $\langle \mathsf{G}, \mathsf{G}', \mathcal{C}\rangle$ is *well-formed* if it satisfies the following predicate:

$$\begin{array}{l}\mathit{WellFormed}(\mathsf{G}, \mathsf{G}', \mathcal{C}) \triangleq \\ \quad \mathit{WellFormed}(\mathsf{G}) \wedge \\ \quad \forall c \in \mathcal{C}.\ \mathsf{G}.\mathsf{tid}(c) = \mathsf{G}'.\mathsf{tid}(c) \wedge \\ \quad \forall c \in \mathcal{C}.\ \mathsf{G}.\mathsf{lab}(c) = \mathsf{G}'.\mathsf{lab}(c) \wedge \\ \quad [\mathsf{G}.\mathcal{C}];\mathsf{G}'.\mathsf{po};[\mathsf{G}.\mathcal{C}] \subseteq \mathsf{G}.\mathsf{po} \wedge \\ \quad [\mathsf{G}.\mathcal{C}];\mathsf{G}'.\mathsf{rf};[\mathsf{G}.\mathcal{C}] \subseteq \mathsf{G}.\mathsf{rf} \wedge \\ \quad [\mathsf{G}.\mathcal{C}];\mathsf{G}'.\mathsf{mo};[\mathsf{G}.\mathcal{C}] \subseteq \mathsf{G}.\mathsf{mo} \wedge \\ \quad \mathsf{G}.\mathsf{R} \subseteq \mathsf{codom}(\mathsf{G}.\mathsf{rf}) \cup \mathcal{C} \\ \quad \mathsf{G}'.\mathsf{R} \cap \mathcal{C} \subseteq \mathsf{codom}([\mathcal{C}];\mathsf{G}'.\mathsf{rf}) \wedge\end{array}$$

where $\mathsf{G}.\mathcal{C} \triangleq \mathsf{G}.\mathsf{E} \cap \mathcal{C}$.

Definition 5.0.6 (EmbeddedSubGraph)**.** Given two graphs: G, G'; a set of events $\mathcal{C} \subseteq \mathsf{G}'.\mathsf{E}$ and a partial event mapping function $\mathsf{f} : \mathsf{G}'.\mathsf{E} \rightharpoonup \mathsf{G}.\mathsf{E}$, we say that a subgraph of G' restricted to subset of events $\mathcal{C}$ *is embedded into* graph G under f, if the following conditions are met:

$$
\begin{aligned}
&\mathit{EmbeddedSubGraph}(\mathsf{G}, \mathsf{G}', \mathcal{C}, \mathsf{f}) \triangleq \\
&\quad \mathsf{f} \text{ is injective} \wedge \\
&\quad \forall c \in \mathcal{C}.\ \mathsf{f}(c) \neq \bot \wedge \\
&\quad \forall c \in \mathcal{C}.\ \mathsf{G}.\mathsf{tid}(\mathsf{f}(c)) = \mathsf{G}'.\mathsf{tid}(c) \wedge \\
&\quad \forall c \in \mathcal{C}.\ \mathsf{G}.\mathsf{lab}(\mathsf{f}(c)) = \mathsf{G}'.\mathsf{lab}(c) \wedge \\
&\quad \mathsf{f} \uparrow ([\mathcal{C}]; \mathsf{G}'.\mathtt{rpo}; [\mathcal{C}]) = [\mathsf{f} \uparrow \mathcal{C}]; \mathsf{G}.\mathtt{rpo}; [\mathsf{f} \uparrow \mathcal{C}] \wedge \\
&\quad \mathsf{f} \uparrow ([\mathcal{C}]; \mathsf{G}'.\mathsf{rf}; [\mathcal{C}]) = [\mathsf{f} \uparrow \mathcal{C}]; \mathsf{G}.\mathsf{rf}; [\mathsf{f} \uparrow \mathcal{C}] \wedge \\
&\quad \mathsf{f} \uparrow ([\mathcal{C}]; \mathsf{G}'.\mathsf{mo}; [\mathcal{C}]) = [\mathsf{f} \uparrow \mathcal{C}]; \mathsf{G}.\mathsf{mo}; [\mathsf{f} \uparrow \mathcal{C}]
\end{aligned}
$$

Definition 5.0.7 (StableUncommittedReads)**.** Given a graph G' and a set of its committed events $\mathcal{C} \subseteq \mathsf{G}'.\mathsf{E}$, let $\mathcal{U} \triangleq \mathsf{G}'.\mathsf{E} \backslash \mathcal{C}$ be a set of uncommitted events. We say that *uncommitted reads are stable*, denoted as $\mathit{StableUncommitedReads}(\mathsf{G}', \mathcal{C})$, if there exists a partial order relation $\underset{\mathsf{tid}}{\prec}$, which orders events from different threads, that satisfies the following conditions:

- $\underset{\mathsf{tid}}{=^?}; \underset{\mathsf{tid}}{\prec}; \underset{\mathsf{tid}}{=^?} \subseteq \underset{\mathsf{tid}}{\prec}$ — the given order is closed with respect to same-thread events;
- $\mathsf{rf}; [\mathcal{U}] \subseteq \underset{\mathsf{tid}}{\prec}$ — uncommitted reads are aligned with the given order.

5.1 Execution Step

When executing a graph G with a set of committed events C, we perform in-order execution of instructions by adding a new event e to G, resulting in G'. G' is checked for consistency and possibly discarded.

If $e \in R \wedge e \notin C$ (e is an uncommitted read), it is required to read from an existing write event, preventing the formation of $\mathsf{po} \cup \mathsf{rf}$ cycles. If $C = \emptyset$, then G' will be an RC11 consistent execution.

If $e \in R \wedge e \in C$ (e is a committed read), it can temporarily read from "nowhere" until a corresponding committed write is added to G. A committed read can only read from a committed write and should read from the same thread as in the original graph.

5.2 Re-Execution Step

Once a graph G is complete, we choose a subset of determined events D from the events of G and a set of committed events C. Committed events will be preserved during re-execution.

We then re-execute starting from the configuration $\langle \mathsf{G}|_{\mathcal{D}}, \mathsf{G}', \mathcal{C} \rangle$ and arrive at the resulting graph G', checking its consistency at the end.

Finally, G' has three additional constraints:

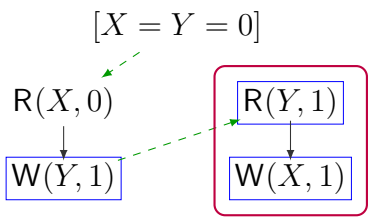
1. *EmbeddedSubGraph* ensures that the subgraph of committed events is contained in the original graph G;
2. *CausalRestriction* assures that the causality relation $\mathsf{G}'.\mathsf{porf}$ has an expected shape;
3. *StableUncommitedReads* restricts the reads-from relation $\mathsf{G}'.\mathsf{rf}$ for uncommitted reads.

5.3 Step-by-Step Example

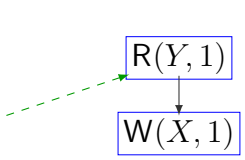
Let us demonstrate the semantics of XMM on the LB litmus test:

$$\begin{array}{l|\!|l} a := X & b := Y \\ Y := 1 & X := b \end{array}$$

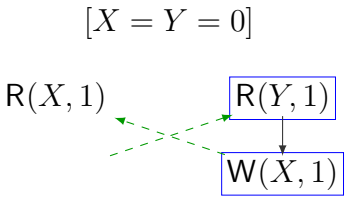
We begin the *Execution* step with an empty graph G and an empty set of committed events. One of the executions we can obtain is the following:



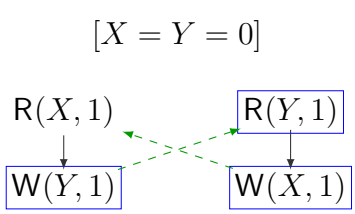
We can now choose the set of committed events C highlighted in blue and the set of determined events D highlighted in purple. With C and D, we can begin re-execution:



Next, we add the read event corresponding to $a := X$ in thread 1. We choose its rf edge to come from $\mathsf{W}(X, 1)$.



Lastly, the committed $\mathsf{W}(Y, 1)$ is added to thread 1:



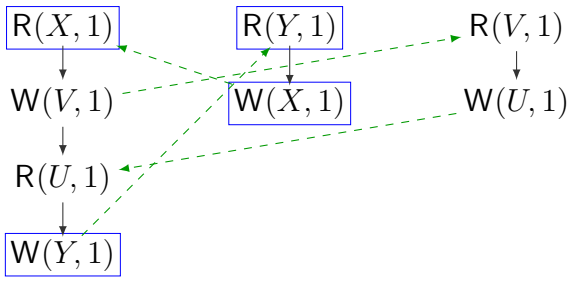
This graph G' satisfies the constraints: *Consistent*, *EmbeddedSubGraph*, and *StableUncommittedReads*, therefore, it is valid according to XMM.

5.4 Stable Uncommitted Reads Constraint

The XMM model defines an additional constraint on graphs, called *StableUncommittedReads(G, G')* (Definition 5.0.7). This constraint is required for the reordering transformations to be sound.

Take, for example, the following graph, where committed events are highlighted in blue. This graph is not XMM consistent because its uncommitted events are not stable: from $\mathsf{W}(V,1)$ in thread 1, there is an rf edge to $\mathsf{R}(V,1)$ in thread 3, and from $\mathsf{W}(W,1)$ in thread three we can go back to thread one via the rf edge to $\mathsf{R}(U,1)$.

$$[U = V = X = Y = 0]$$



This constraint is needed because graphs that do not satisfy this constraint may get different XMM consistent executions if some of their events are reordered:

$$\begin{array}{l||l||l}
a := X & & \\
\textbf{if}\,(a == 0)\,\{ & & \\
\quad Y := 1 & & \\
\} & & \\
\textbf{else}\{ & r1 := Y & r2 := V \\
\quad V := 1 & X := r1 & U := r2 \\
\quad b := U & & \\
\quad Y := b & & \\
\} & &
\end{array}$$

Suppose that XMM did not have the *StableUncommitedReads*$(\mathsf{G}', \mathcal{C})$ constraint. Then, the execution shown above would be a consistent XMM execution of this program. However, if we reordered the statements $V := 1$ and $b := U$ in thread 1, that execution would not be a consistent XMM execution anymore.

5.5 Load Buffering Race Freedom

In [34], the authors proved that XMM satisfies the *Load Buffering Race Freedom* RC11 property. This property guarantees that, for a program with no load buffering races, its behavior under XMM is the same as under RC11.

Theorem 1 (LBRF-RC11)**.** XMM provides load buffering race freedom guarantee with respect to the RC11 model:

$$\mathsf{XMM} \in \mathsf{LBRF}(\mathsf{RC11})$$

5.6 Supported Transformations

XMM's lax rules allow the compiler to perform several optimizations. The proof of the correctness of the optimizations listed below can be found in [34].

- *Store-Store, Store-Load, Load-Store, Load-Load Reorderings*: the compiler is free to reorder any two adjacent events as long as they access different memory locations and there is no fence between them.
- *Load-Load, Store-Load, Store-Store Elimination*: some loads and stores can be eliminated. Local variables are used instead, or in the case of store-store elimination, the first store is dropped completely.
- *Sequentialization*: two threads can be merged into one by appending one to the other.

Chapter 6

The GenMC-XMM Model Checker

In this chapter, we present GenMC-XMM, a new model checker designed for the XMM relaxed memory model [34]. XMM allows a wide range of compiler optimizations, and it lends itself to be easily verified by a model checker. We built GenMC-XMM on top of GenMC$_{RC11}$ (Chapter 3). While GenMC was designed to be parametric in the choice of the underlying memory model, extending GenMC to work for XMM required extensive modifications because GenMC assumes that the underlying model is po + rf acyclic, which XMM is not. The main principle behind GenMC-XMM is to let GenMC find the RC11 executions, look for *load buffering races*, and explore cyclic executions only when such races are found. This is possible because of the LBRF$_{\mathsf{RC11}}$ property of XMM (Section 5.5) that states that cyclic executions are present only if the program has one or more load buffering races. We represent the transitions between RC11 graphs and XMM graphs in Figure 6.1.

Figure 6.1: Transitions between RC11 and XMM graphs

The main issue we encountered while designing GenMC-XMM is that the XMM model specifies that committed and determined events can be chosen arbitrarily. This translates to trying all subsets of events in the current graph. The resulting algorithmic time complexity is $O(2^n)$. We discarded this approach because it would not have scaled to graphs with hundreds or thousands of events that current state-of-the-art model checkers can handle. Instead, we opted for a heuristic approach driven by the patterns we found in test programs and our goal of facilitating the proof of soundness. Our tests found that this sound but incomplete approach is enough to find all executions in 71 out of 73 tests (Section 7.1).

We provide a detailed pseudocode representation of GenMC-XMM in Algorithm 2. The pseudocode uses a special notation $a \leftarrow A$ to indicate that a can be any element of set A. This notation is inspired by the non-deterministic "do" notation of some functional languages. Regular variable assignment uses notation $x := 2 + 2$.

Algorithm 1: Helper Functions

G.poSuffix(e) = $[\{e\}]; \mathsf{G.po}^*$
G.porfPrefix(e) = $\mathsf{G.po}^* \cup \mathsf{G.rf}^*; [\{e\}]$
G.reads() = R
G.writes() = W
G.lbRaces() = $\{(r, w) \mid \mathsf{G.LBRace}(r, w)\}$
G.numThreads() = $\mid \mathsf{TID} \mid$
r.rf = $\mathsf{dom}(\mathsf{G.rf}; [\{r\}])$

Algorithm 2: GenMC-XMM

```
// In this pseudocode:
// a ← A indicates that we non-deterministically pick any element a from set A
// x := 2+2 is a normal variable assignment
```

```
1  func visit(P, G):
2    return set of graphs that GenMC can construct by adding events to G, following program P
3
4  func matchRfEdges(G_XMM, G_original):
5    r_⊥ ← G_XMM.reads().filter(r_⊥.rf = ⊥)
6    r_original := G_original.get(r_⊥)
7    w ← G_XMM.writes().filter(w.value = r_⊥.value ∧ w.loc = r_⊥.loc ∧ w.tid = r_original.rf.tid)
8    return matchRfEdges(G_XMM.changeRf(r_⊥, w), G_original)
9
10 func visitLBRaces(P, G): // called every time an RC11 graph is found
11   ⟨racyW, racyR⟩ ← G.lbRaces()
12
13   D := G \ G.poSuffix(racyR)
14   G_XMM ← visit (P, D)
15   G_XMM ← matchRfEdges (G_XMM, G)
16   visitXmmGraph (P, G_XMM)
17
     // revisit reads outside porf cycle
18   D_rOutCycle := G_XMM.porfPrefix(racyR)
19   G_rOutCycle ← visit (P, D_rOutCycle)
20   visitXmmGraph (P, G_rOutCycle)
21
     // revisit reads in the thread of racyR
22   r ← G_XMM.poPrefix(racyR).reads
23   D_racyRThread := G_XMM \ G_XMM.poSuffix(r)
24   G_racyRThread ← visit (P, D_racyRThread)
25   G_racyRThread ← matchRfEdges (G_racyRThread, G_XMM)
26   visitXmmGraph (P, G_racyRThread)
27
28 func visitXmmGraph(P, G):
29   if ¬ G.consistent() ∨ ¬ G.areUncommittedReadsStable() ∨ ¬ G.embeddedSubGraph() ∨
       G.duplicate():
30     return
31   output (G)
32   visitLBRaces (P, G)
```

6.1 Algorithm Outline

The GenMC-XMM algorithm is divided into three phases:

1. Create a cycle from an RC11 graph (lines 11-16) (Section 6.2).
2. Revisit reads outside the generated cycle (lines 18-20) (Section 6.3).
3. Revisit reads in the thread of the LB racy read (lines 22-26) (Section 6.4).

We now explain the pseudocode of each phase.

Phase 1:

1. GenMC-XMM exploits the $\text{LBRF}_{\text{RC11}}$ property of XMM: we only construct cyclic executions if we encounter an LB race in a RC11 execution. When an RC11 execution is complete, *visitLBRaces* is called. Argument P represents the current program, and G the RC11 graph.
2. At lines 11-13, we scan the graph for LB races by applying Definition 2.4.2. For each LB race found between a read *racyR* and a write *racyW*, we restrict the original graph G, removing all the events in the po-suffix of *racyR* (including itself), resulting in a restricted graph D.
3. At lines 14-15, we re-execute D by calling *visit(P, D)*. *visit* returns the set of all graphs that $\text{GenMC}_{\text{RC11}}$ can construct from D following program P. We non-deterministically pick any graph from that set and call it G_{XMM}. At this point, G_{XMM} can have some reads that do not have an rf edge ($\text{R}_{\perp}$) because the graph restriction could have removed the corresponding write. We use helper function *matchRfEdges* to complete G_{XMM} with the missing rf edges.
4. At line 16, G_{XMM} is complete. We call *visitXmmGraph*, which outputs G_{XMM} if it is consistent (Section 6.6), not a duplicate (Section 6.7), and the *stableUncommittedReads* (Section 6.8) and *embeddedSubGraph* (Section 6.9) constraints are satisfied. Lastly, *visitXmmGraph* calls *visitLBRaces* to check if G_{XMM} has any new LB races we can use to derive a new graph.

Phase 2:

1. At lines 18-20, we restrict G_{XMM}. We remove all events not in the porf-prefix of racyR and re-execute by calling *visit*. Because we maintain the porf-prefix of racyR, and thus the cycle, the re-executed graph $\text{G}_{\text{rOutCycle}}$ does not have any $\text{R}_{\perp}$, so we do not need to call *matchRfEdges*. We call *visitXmmGraph*, which outputs $\text{G}_{\text{rOutCycle}}$ and looks for new LB races in it.

Phase 3:

1. At lines 22-24, we non-deterministically pick a read from the po-prefix of racyR and call it r. We restrict G_{XMM} once more by removing the po-suffix of r, and we re-execute by calling *visit*. The re-executed graph $\text{G}_{\text{racyRThread}}$ can have $\text{R}_{\perp}$, so we add the missing rf edges with *matchRfEdges*. We call *visitXmmGraph*, which outputs $\text{G}_{\text{racyRThread}}$ and looks for new LB races in it.

matchRfEdges helper function:

1. The *matchRfEdges* helper function takes as arguments two graphs: G_{XMM} is the graph with $\text{R}_{\perp}$, and $\text{G}_{\text{original}}$ is the graph from which we derived G_{XMM}. At line 5, we non-deterministically pick a read that misses its rf edge from G_{XMM} and call it $\text{r}_{\perp}$. At line 6,

we find the equivalent of $\mathrm{r}_\perp$ in $\mathrm{G}_{\text{original}}$ and call it $\mathrm{r}_{\text{original}}$. The only difference between $\mathrm{r}_\perp$ and $\mathrm{r}_{\text{original}}$ is that, unlike the former, the latter has an rf edge. At line 7, we non-deterministically pick from G_{XMM} a same-value, same-location write w in the thread from which $\mathrm{r}_{\text{original}}$ reads. Lastly, we add an rf edge between $\mathrm{r}_\perp$ and w in G_{XMM} and call *matchRfEdges* recursively to complete the rest of the $\mathrm{R}_\perp$.

In our algorithm, the sets of committed events $\mathcal{C}$ and determined events $\mathcal{D}$ are not arbitrary like in the XMM semantics, but they are chosen heuristically:

$$\mathcal{C}_{\text{GenMC-XMM}} = G_{\text{restricted}}.\mathsf{E} \cup \{w \in \mathsf{G}' \mid w \text{ is a write matched to some } R_\perp\}$$

Events are committed if they are part of the graph after it is restricted or matched to a $R_\perp$ by the *matchRfEdges* function. The intuition behind this is that events in the restricted graph and matched writes are, by definition, both in G and G′, like committed events in XMM semantics.

$$\mathcal{D}_{\text{GenMC-XMM}} = G_{\text{restricted}}.\mathsf{E}$$

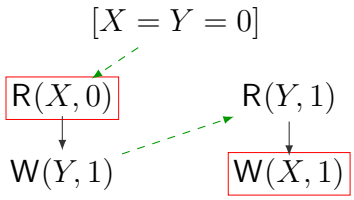
Events are determined if they are part of the graph after it is restricted. The intuition behind this is the restricted graph is the starting point of re-execution in GenMC-XMM, like determined events in XMM semantics.

6.2 Step-by-Step Example

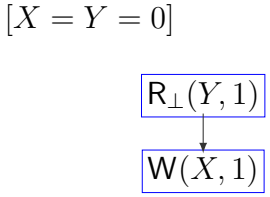
Let us run the GenMC-XMM algorithm (Algorithm 2) on the LB litmus test shown below:

$$\begin{array}{l|\!|l} a := X & b := Y \\ Y := 1 & X := b \end{array}$$

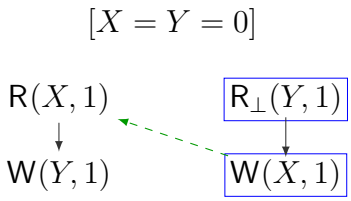
The *visitLBRaces* function is called with the complete RC11 execution found by $\text{GenMC}_{\mathsf{RC11}}$ shown below. We enumerate the LB races (line 11) and find the pair of racy events highlighted in red. We do not consider $\langle \mathsf{R}(Y,1), \mathsf{W}(Y,1)\rangle$ as an LB race because an rf edge already links them.



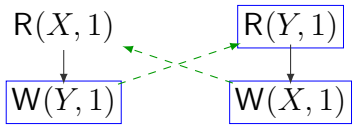
We remove the po-suffix of $\mathsf{R}(X,0)$, resulting in a restricted graph D shown below (line 13). Notice that $\mathsf{R}(Y,1)$ is now an $\mathrm{R}_\perp$ because it misses its rf edge. We highlighted in blue the "XMM semantics committed" events.



At line 14, we call *visit* with the previous graph as argument. One of the graphs returned by *visit* is the following (variable G_{XMM} in the pseudocode):



At line 15, we call *matchRfEdges* with the previous graph as argument. *matchRfEdges* finds $\mathsf{R}_\perp(Y,1)$ in thread 2 and compatible write $\mathsf{W}(Y,1)$ in thread 1, so it adds an rf edge between them. $\mathsf{W}(Y,1)$ is considered committed from now on because it was matched to a committed read. The resulting $\mathrm{G}_{\mathrm{XMM}}$ is the following:



Since this graph is consistent (Section 6.6), it is not a duplicate (Section 6.7), *stableUncommittedReads* (Section 6.8) holds, and *embeddedSubGraph* (Section 6.9) holds, *visitXmmGraph* outputs it.

In lines 18-26 (Algorithm 2), we further restrict $\mathrm{G}_{\mathrm{XMM}}$, but in this example, we do not find any further XMM graphs.

The rest of the execution graphs are acyclic and are obtained from GenMC_{RC11} as shown in Section 3.2.

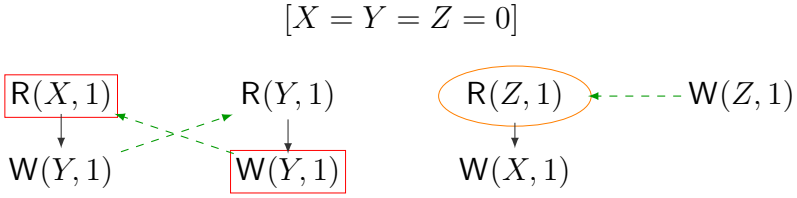
6.3 Revisiting Reads Outside the Cycle

After obtaining an XMM execution, it is possible that some reads that are not in the cycle or its porf-prefix can be revisited to read-from happens-before (hb) previous writes. We achieve this by restricting the graph, keeping all events that are part of the LB cycle and its porf-prefix, and removing everything else (lines 18-20 in Algorithm 2). To derive new graphs, we re-execute the restricted graph.

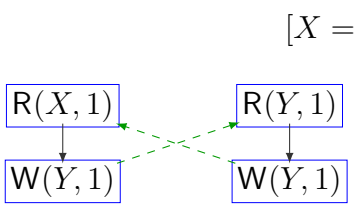
Let us take as an example Java causality test 10:

$$\begin{array}{l} r1 := X \\ \textbf{if}\,(r1 == 1)\ \{ \\ \quad Y := 1 \\ \} \end{array} \;\Big\|\; \begin{array}{l} r2 := Y \\ \textbf{if}\,(r2 == 1)\ \{ \\ \quad X := r2 \\ \} \end{array} \;\Big\|\; \begin{array}{l} r3 := Z \\ \textbf{if}\,(r3 == 1)\ \{ \\ \quad X := r3 \\ \} \end{array} \;\Big\|\; Z := 1$$

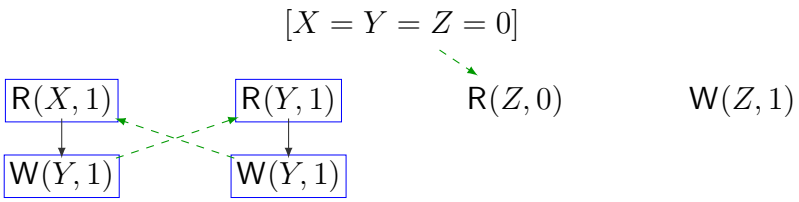
This is the first XMM consistent execution explored by GenMC-XMM:



We want to revisit $\mathsf{R}(Z,1)$ to read from the initialization write $Z=0$ instead of $\mathsf{W}(Z,1)$, so we remove all events that are not part of the porf-prefix of the LB racy read $\mathsf{R}(X,1)$:



By re-executing this graph, we obtain the desired graph shown below. We highlighted in blue the committed events according to XMM semantics.



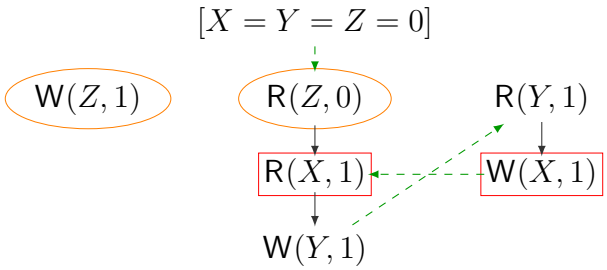
6.4 Revisiting Reads in the LB Racy Read Thread

A read in the thread of the LB racy read may get a new potential rf edge after the creation of a cycle in the graph. Let us call *racyR* the LB racy read and r the read in the thread of *racyR* that we wish to revisit. We restrict the graph, removing the po-suffix of r. After restriction, some reads become $R_\perp$. We re-execute the restricted graph by calling *visit* (lines 22-24 in Algorithm 2). On line 25, we call *matchRfEdges*, which completes the missing rf edges.

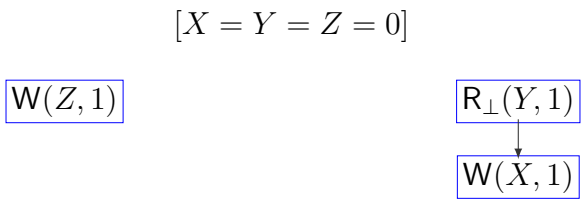
Take, for example, this test case taken from [28]:

$$Z := 1 \;\Big\|\; \begin{array}{l} r := Z \\ s := X \\ \textbf{if}\,(r == 0 \vee s == 1)\ \{ \\ \quad Y := 1 \\ \} \end{array} \;\Big\|\; \begin{array}{l} a := Y \\ X := a \end{array}$$

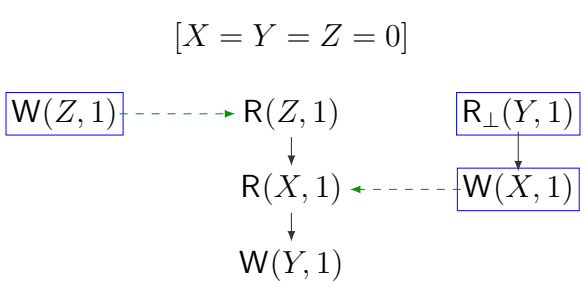
GenMC-XMM produces the following cyclic execution:



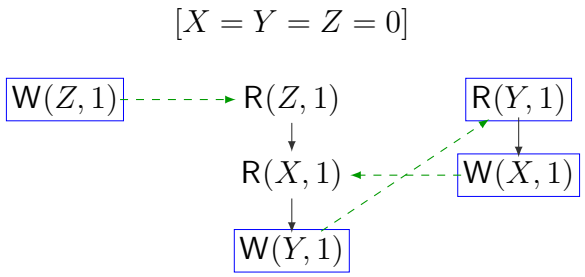
Now that we obtained $s := 1$ through the cycle, $r == 0$ is not necessary anymore for the *if* statement to be true, so we can change the rf edge of $\mathsf{R}(Z,0)$ to read from $\mathsf{W}(Z,1)$ without breaking the cycle. We restrict the graph, removing the po-suffix of $\mathsf{R}(Z,0)$, and obtain:



The *visit* function at line 24 with the previous graph as argument produces the following graph:



At line 25, we call *matchRfEdges*, which finds $\mathsf{R}_\perp(Y, 1)$ and compatible write $\mathsf{W}(Y, 1)$, and adds an rf edge between them. The committed events according to XMM semantics are highlighted in blue.



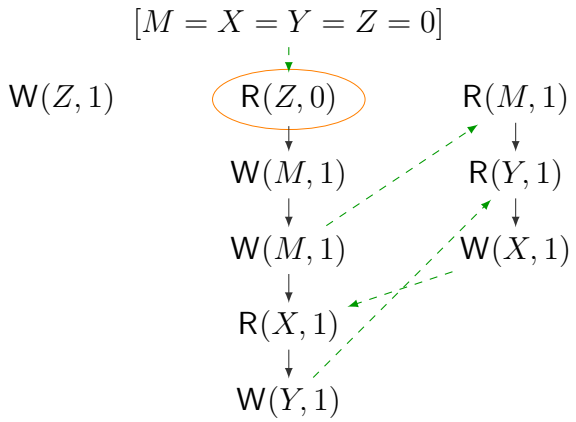
6.5 $R_\perp$ rf matching

In this section, we show an example where the *matchRfEdges* function (lines 4-8 in Algorithm 2) finds multiple writes compatible with a $\mathrm{R}_\perp$. *matchRfEdges* returns a set containing one graph for each option.

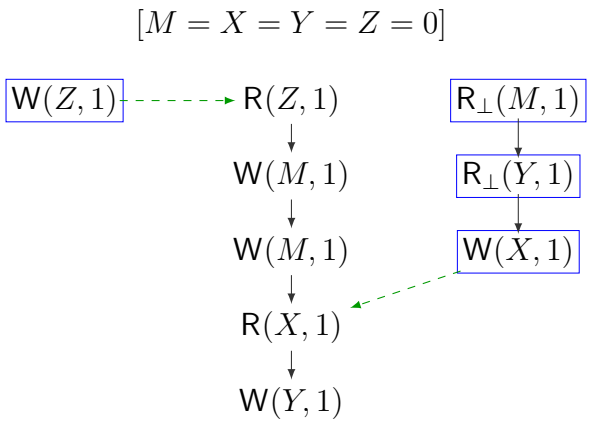
Let us take the following test:

$$Z := 1 \;\Big\|\; \begin{array}{l} r1 := Z \\ M := 1 \\ M := 1 \\ r2 := X \\ \textbf{if}\,(r1 == 0 \vee r2 == 1)\;\{ \\ \quad Y := 1 \\ \} \end{array} \;\Big\|\; \begin{array}{l} r3 := M \\ r4 := Y \\ X := r4 \end{array}$$

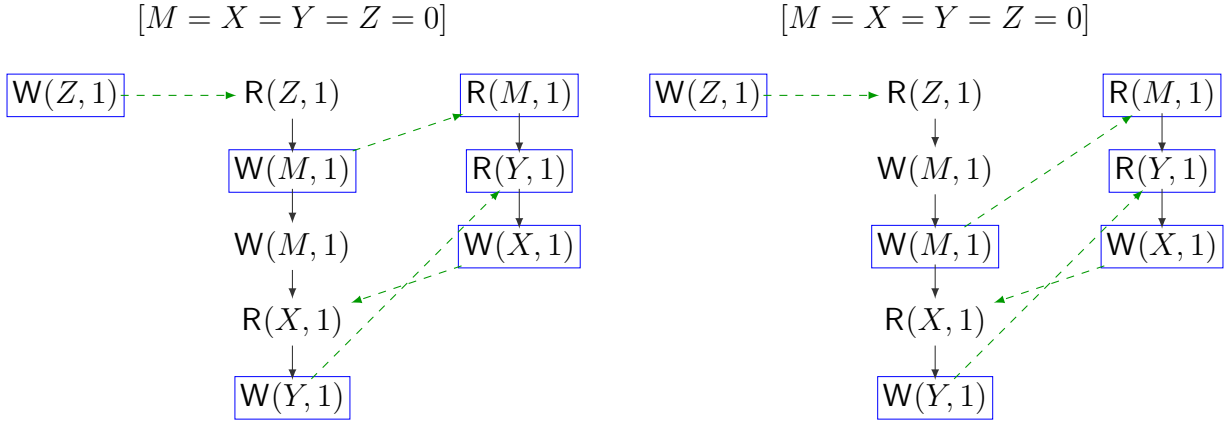
GenMC-XMM first generates the following cyclic execution:



When GenMC-XMM revisits $\mathsf{R}(Z, 0)$ to read from $\mathsf{W}(Z, 1)$, it restricts the previous graph, removing the po-suffix of $\mathsf{R}(Z, 0)$. Re-executing the restricted graph results in:



At this point, the *matchRfEdges* function is called with the above graph as $\mathrm{G}_{\mathrm{XMM}}$, and the graph shown above that one as $\mathrm{G}_{\mathrm{original}}$. At line 5, we non-deterministically pick a $\mathrm{R}_{\perp}$. This example has two options: $\mathsf{R}_{\perp}(M, 1)$ and $\mathsf{R}_{\perp}(Y, 1)$. In line 7, we non-deterministically pick a compatible write w. $\mathsf{R}_{\perp}(M, 1)$ is compatible with either one of the two $\mathsf{W}(M, 1)$. $\mathsf{R}_{\perp}(Y, 1)$ is compatible only with $\mathsf{W}(Y, 1)$. Each combination of matches results in a different graph, shown below. Adding a new rf without restricting and re-executing the graph is sufficient since the value read by $\mathsf{R}_{\perp}(M, 1)$ and $\mathsf{R}_{\perp}(Y, 1)$ does not change.



6.6 Consistency

The XMM consistency predicate has three conditions (Definition 5.0.4): Coherence, Atomicity, and SC. GenMC-XMM does not need extra checks for Atomicity and SC because it uses GenMC to construct graphs, and GenMC already provides these two guarantees. For the coherence property, GenMC performs coherence checks progressively as the execution graph gets built. GenMC-XMM cannot easily do the same due to the possible presence of $R_\perp$ in the graph. Instead, it performs a single check when the graph is complete.

Following coherence definition 5.0.4, GenMC-XMM labels a graph G as coherent if there are no two events X and Y in G such that: $X \rightarrow_{\mathsf{hb}} Y \wedge Y \rightarrow_{\mathsf{eco}} X$ (Algorithm 3).

Since all pairs of events need to be enumerated, this coherence check takes $O(n^2)$ time complexity, considering n as the number of events in graph G. This approach on complete graphs is not as efficient as the step-by-step approach taken by GenMC since the latter discards graphs earlier while they are still being constructed. In the future, it would be interesting to design a step-by-step approach for GenMC-XMM and measure the performance speed-up.

Algorithm 3: Consistency Check

```
1 func isCoherent(Graph g):
2     for e1 ∈ g.events():
3         for e2 ∈ g.events():
4             if e2.hb.contains(e1) ∧ e1.eco.contains(e2):
5                 return false
6     return true
7 func isConsistent(Graph g):
      // the Atomicity and SC constraints are guaranteed by GenMC
8     return isCoherent(g)
```

6.7 Duplicates

As explained in GenMC's paper [21], *forward* revisits of a read R do not lead to duplication because only events added after R are removed from the graph. *Backward* revisits, on the other hand, can lead to duplicate executions. To tackle this issue, GenMC$_{RC11}$ marks explored backward options as *visited* and does not consider them in the future. This design allows GenMC to never re-execute a graph that would result in a duplicate.

Due to cyclic graphs, GenMC-XMM cannot use the same strategy. A naive approach would be to record which LB races have been explored, but different LB races can lead to the same resulting graph. Instead, GenMC-XMM memorizes the entire graph in a hash set and discards it if it is already present (Algorithm 4). This strategy is not as efficient as GenMC's because it does not allow to know whether a graph is a duplicate until it is fully re-executed, and it adds the overhead of hashing the graphs. In the future, it may be interesting to research a more efficient solution to prevent duplicates and compare the performance speed-up.

Algorithm 4: Duplicate Graph

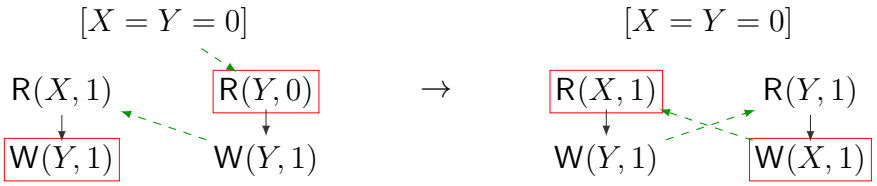
```
1 func isDuplicate(G):
    // visitedGraphs is a global set of graphs
2   if G ∈ visitedGraphs:
3     return true
4   else:
5     visitedGraphs := visitedGraphs ∪ {G}
6     return false
```

To show how two different LB races lead to the same graph, take, for example, the simplest LB test:

$$\begin{array}{l|\!|l} a := X & b := Y \\ Y := 1 & X := 1 \end{array}$$

From the RC11 graph on the left, we get the cyclic execution on the right.

But from this different RC11 graph, with a different LB race, we obtain the same cyclic graph:



GenMC-XMM even creates a second duplicate execution because the algorithm checks whether reads outside the cycle can be revisited (lines 18-20 in Algorithm 2). The graph is restricted to contain only events in the cycle, resulting in the same graph in this example.

For this particular example, GenMC-XMM explores four valid unique executions and discards two duplicates.

6.8 Stable Uncommitted Reads

The XMM semantics specify that any XMM consistent graph should satisfy the *StableUncommittedReads* constraint (Definition 5.0.7).

Following the definition, we have implemented the constraint as shown in Algorithm 5. The algorithm constructs a ”data flow” graph with one node for each tid and edges between them if there is an uncommitted read with an rf edge between them. The algorithm returns true if this graph is acyclic.

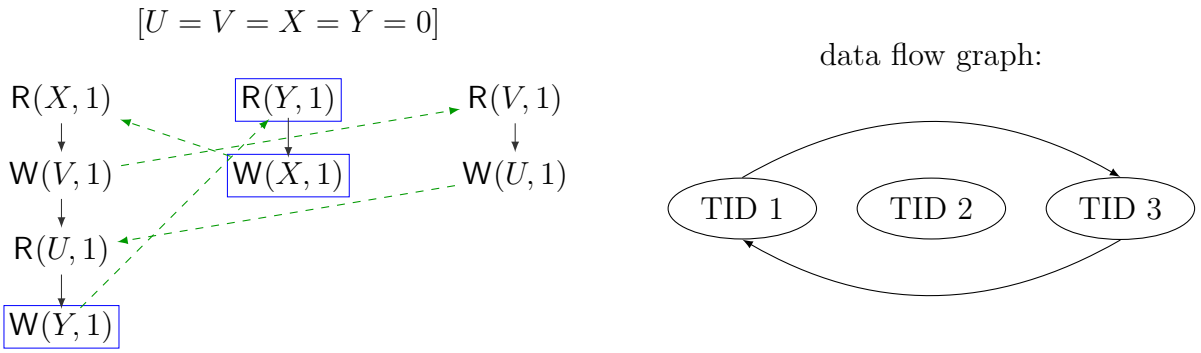
Algorithm 5: Stable Uncommitted Reads

```
1  func constructDataflow(graph, uncommittedReads):
2      dataFlow := Graph()
3      for uncommittedRead ∈ uncommittedReads:
4          if uncommittedRead.rf.tid ≠ uncommittedRead.tid:
5              dataFlow.addEdge(uncommittedRead.rf.tid, uncommittedRead.tid)
6      return dataFlow
7
8  func areUncommittedReadsStable(G, uncommittedReads):
9      dataFlow := constructDataflow(G, uncommittedReads)
10     return isAcyclic(dataFlow)
```

We demonstrate how this algorithm works on the following test program:

$$\begin{array}{l} \textbf{if}\,(X == 0)\,\{ \\ \quad Y := 1 \\ \} \\ \textbf{else}\{ \\ \quad V := 1 \\ \quad Y := U \\ \} \end{array} \;\Big\|\; X := Y \;\Big\|\; U := V$$

In the graph on the left below, committed events are highlighted in blue, and the others are uncommitted. The graph on the right is the "data flow" graph corresponding to the execution graph on the left. Since it has a cycle, this execution is not allowed by XMM and is discarded by GenMC-XMM.



6.9 Embedded Sub-Graph Check

The XMM semantics impose the *EmbeddedSubGraph* constraint (Definition 5.0.6) on a graph G' obtained by re-executing G with a set of committed events $\mathcal{C}$, and event mapping function f.

All conditions of the *EmbeddedSubGraph* constraint are satisfied by GenMC-XMM without the need for extra checks (as proved in Section 6.10), except the following two:

1. $\mathsf{f}\uparrow([\mathcal{C}];\mathsf{G}'.\mathsf{mo};[\mathcal{C}]) = [\mathsf{f}\uparrow\mathcal{C}];\mathsf{G}.\mathsf{mo};[\mathsf{f}\uparrow\mathcal{C}]$
2. $\mathsf{f}\uparrow([\mathcal{C}];\mathsf{G}'.\mathtt{rpo};[\mathcal{C}]) = [\mathsf{f}\uparrow\mathcal{C}];\mathsf{G}.\mathtt{rpo};[\mathsf{f}\uparrow\mathcal{C}]$

We have added an extra check in GenMC-XMM that discards graphs that do not satisfy the above two conditions (Algorithm 6).

Condition 1 is checked at lines 2-7:

1. We non-deterministically get a committed write w from the po-suffix of the LB racy read in the derived graph.
2. We get an equivalent write w_{original} in the po-suffix of the LB racy read in the original graph.
3. Calculate the mo-prefixes of w and w_{original}, removing events in the po-suffix of the LB racy read, and check that they are equal.

Condition 2 is checked at lines 9-11:

1. We non-deterministically get an event e in the po-path between the LB racy read, and the committed write w.
2. We check that e no new rpo edges between e and w are created.

Algorithm 6: Embedded Subgraph Check

```
1  func createsRpoEdge(e, w):
2      if e.order = Acquire:
3          return true
4      if e.order = Release:
5          if e.type = Fence:
6              return true
7          if e.type = Write ∧ e.loc = w.loc:
8              return true
9      return false
10
11 func embeddedSubgraph(G, G_original, racyR):
       // check mo preserved
12     w ← G.poSuffix(racyR).writes().filter(w → w.isCommitted())
13     w_original := G_original.poSuffix(racyR).find(w)
14     moPrefix := G.moPrefix(w).filter(p → p ∉ G.poSuffix(racyR)
15     moPrefix_original := G_original.moPrefix(w_original).filter(p → p ∉ G_original.poSuffix(racyR)
16     if moPrefix ≠ moPrefix_original:
17         return false
18
       // check rpo preserved
19     e ← G.poSuffix(racyR) ∩ G.poPrefix(w)
20     if createsRpoEdge(e, w):
21         return false
22
23     return true
```

6.10 Soundness of GenMC-XMM

The XMM model uses a notion of committed events, while GenMC-XMM uses a notion of restricted graphs and $R_\perp$ matching. So, how are they related? Is GenMC-XMM sound?

GenMC-XMM cannot use a non-deterministic approach when selecting the set of committed events $\mathcal{C}$ like XMM prescribes. Trying all subsets of events would only work for small tests since the time complexity would be $O(2^{|C|})$ where $\mathcal{C}$ is the set of events in a graph. Instead, GenMC-XMM assumes that all the events in a restricted graph $G_{\text{restricted}}$ and the writes matched to a $R_\perp$

are committed. The same reasoning applies to the set of determined events $\mathcal{D}$. GenMC-XMM always chooses $\mathcal{D}$ equal to all events in the restricted graph $G_{\text{restricted}}$.

$$\mathcal{C}_{\text{GenMC-XMM}} = G_{\text{restricted}}.\mathsf{E} \cup \{w \in \mathsf{G}' \mid w \text{ is a write matched to some } R_\perp\}$$
$$\mathcal{D}_{\text{GenMC-XMM}} = G_{\text{restricted}}.\mathsf{E}$$

Theorem 2.

GenMC-XMM is sound.

Proof.

GenMC-XMM is sound if, given the above definitions of $\mathcal{C}_{\text{GenMC-XMM}}$ and $\mathcal{D}_{\text{GenMC-XMM}}$, when GenMC-XMM derives a graph G' from G, the following properties hold:

- $\mathit{WellFormed}(\mathsf{G}|_{\mathcal{D}_{\text{GenMC-XMM}}}, \mathsf{G}', \mathcal{C}_{\text{GenMC-XMM}})$
- $\mathit{WellFormed}(\mathsf{G}_{i+1}, \mathsf{G}', \mathcal{C}_{\text{GenMC-XMM}})$
- $\mathit{EmbeddedSubGraph}(\mathsf{G}, \mathsf{G}', \mathcal{C}_{\text{GenMC-XMM}}, \mathsf{f})$
- $\mathit{StableUncommitedReads}(\mathsf{G}', \mathcal{C}_{\text{GenMC-XMM}})$
- $\mathit{Consistent}(\mathsf{G}')$

Each of the above properties are proved to hold in Theorems 4 to 8. □

Theorem 3.

Whenever GenMC-XMM adds an event to a graph, the resulting graph is *WellFormed* (see Definition 5.0.2).

Proof.

GenMC-XMM uses GenMC to add events to graphs. By design, GenMC ensures that the resulting graph is *WellFormed*. Since GenMC-XMM starts from an empty graph, which is trivially *WellFormed*, all graphs explored by GenMC-XMM and their prefixes are also *WellFormed*. □

Theorem 4.

$$\frac{\mathcal{C} = \mathcal{C}_{\text{GenMC-XMM}} \qquad \mathcal{D} = \mathcal{D}_{\text{GenMC-XMM}} \qquad \mathsf{G}|_{\mathcal{D}} \xrightarrow[\texttt{GenMC-XMM}]{(f,C)} \mathsf{G}'}{\mathit{WellFormed}(\mathsf{G}|_{\mathcal{D}}, \mathsf{G}', \mathcal{C})}$$

Proof.

We show that $\mathit{WellFormed}(\mathsf{G}|_{\mathcal{D}}, \mathsf{G}', \mathcal{C})$ holds by unfolding Definition 5.0.5 and showing that each of its conditions holds:

- $\mathit{WellFormed}(\mathsf{G}|_{\mathcal{D}})$ follows from Theorem 3 because $\mathsf{G}|_{\mathcal{D}}$ is a prefix of G'.
- $\forall c \in \mathcal{C}.\ \mathsf{G}|_{\mathcal{D}}.\mathsf{tid}(c) = \mathsf{G}'.\mathsf{tid}(c)$ because $\mathsf{G}|_{\mathcal{D}}$ is a prefix of G'
- $\forall c \in \mathcal{C}.\ \mathsf{G}|_{\mathcal{D}}.\mathsf{lab}(c) = \mathsf{G}'.\mathsf{lab}(c)$ because $\mathsf{G}|_{\mathcal{D}}$ is a prefix of G'
- $[\mathsf{G}|_{\mathcal{D}}.\mathcal{C}]; \mathsf{G}'.\mathsf{po}; [\mathsf{G}|_{\mathcal{D}}.\mathcal{C}] \subseteq \mathsf{G}|_{\mathcal{D}}.\mathsf{po}$ because $\mathsf{G}|_{\mathcal{D}}$ is a prefix of G'

- $[\mathsf{G}|_{\mathcal{D}}.\mathcal{C}]; \mathsf{G}'.\mathsf{rf}; [\mathsf{G}|_{\mathcal{D}}.\mathcal{C}] \subseteq \mathsf{G}|_{\mathcal{D}}.\mathsf{rf}$ because $\mathsf{G}|_{\mathcal{D}}$ is a prefix of G'
- $[\mathsf{G}|_{\mathcal{D}}.\mathcal{C}]; \mathsf{G}'.\mathsf{mo}; [\mathsf{G}|_{\mathcal{D}}.\mathcal{C}] \subseteq \mathsf{G}|_{\mathcal{D}}.\mathsf{mo}$ because $\mathsf{G}|_{\mathcal{D}}$ is a prefix of G'
- $\mathsf{G}|_{\mathcal{D}}.\mathsf{R} \subseteq \mathsf{codom}(\mathsf{G}|_{\mathcal{D}}.\mathsf{rf}) \cup \mathcal{C}$ because $\mathcal{D} \subseteq \mathcal{C}$
- $\mathsf{G}'.\mathsf{R} \cap \mathcal{C} \subseteq \mathsf{codom}([\mathcal{C}]; \mathsf{G}'.\mathsf{rf})$ because $\mathcal{D} \subseteq \mathcal{C} \wedge (\forall r \in \mathsf{G}'.\mathsf{R} \cap \mathcal{C}.\ \mathsf{dom}(\mathsf{G}'.\mathsf{rf}; r) \in \mathcal{D} \vee r$ is matched to a committed write)

□

Theorem 5.

$$\frac{\mathsf{G}_i \xrightarrow[\texttt{GenMC-XMM add event}]{(e,l)} \mathsf{G}_{i+1} \qquad \mathcal{C} = \mathcal{C}_{\text{GenMC-XMM}}}{\mathit{WellFormed}(\mathsf{G}_{i+1}, \mathsf{G}', \mathcal{C})}$$

Proof.

We show that $\mathit{WellFormed}(\mathsf{G}_{i+1}, \mathsf{G}', \mathcal{C})$ holds by unfolding Definition 5.0.5 and showing that each of its conditions holds:

- $\mathit{WellFormed}(\mathsf{G}_{i+1})$ follows from Theorem 3 because G_{i+1} is a prefix of G'.
- $\forall c \in \mathcal{C}.\ \mathsf{G}_{i+1}.\mathsf{tid}(c) = \mathsf{G}'.\mathsf{tid}(c)$ because G_{i+1} is a prefix of G'.
- $\forall c \in \mathcal{C}.\ \mathsf{G}_{i+1}.\mathsf{lab}(c) = \mathsf{G}'.\mathsf{lab}(c)$ because G_{i+1} is a prefix of G'.
- $[\mathsf{G}_{i+1}.\mathcal{C}]; \mathsf{G}'.\mathsf{po}; [\mathsf{G}_{i+1}.\mathcal{C}] \subseteq \mathsf{G}_{i+1}.\mathsf{po}$ because G_{i+1} is a prefix of G'.
- $[\mathsf{G}_{i+1}.\mathcal{C}]; \mathsf{G}'.\mathsf{rf}; [\mathsf{G}_{i+1}.\mathcal{C}] \subseteq \mathsf{G}_{i+1}.\mathsf{rf}$ because G_{i+1} is a prefix of G'.
- $[\mathsf{G}_{i+1}.\mathcal{C}]; \mathsf{G}'.\mathsf{mo}; [\mathsf{G}_{i+1}.\mathcal{C}] \subseteq \mathsf{G}_{i+1}.\mathsf{mo}$ because G_{i+1} is a prefix of G'.
- $\mathsf{G}_{i+1}.\mathsf{R} \subseteq \mathsf{codom}(\mathsf{G}_{i+1}.\mathsf{rf}) \cup \mathcal{C}$ because, when a read r is added to G_i to obtain G_{i+1}, GenMC-XMM always creates an rf edge ending in r. Alternatively, if r was not added, it means that $r \in \mathcal{D}$, and thus $r \in \mathcal{C}$.
- $\mathsf{G}'.\mathsf{R} \cap \mathcal{C} \subseteq \mathsf{codom}([\mathcal{C}]; \mathsf{G}'.\mathsf{rf})$ holds, as we showed in the proof of Theorem 4.

□

Theorem 6.

$$\frac{\mathsf{G} \xrightarrow[\texttt{GenMC-XMM}]{(f,C)} \mathsf{G}' \qquad \mathcal{C} = \mathcal{C}_{\text{GenMC-XMM}}}{\mathit{EmbeddedSubGraph}(\mathsf{G}, \mathsf{G}', \mathcal{C}, \mathsf{f})}$$

Proof.

We show that $\mathit{EmbeddedSubGraph}(\mathsf{G}, \mathsf{G}', \mathcal{C}, \mathsf{f})$ holds by unfolding Definition 5.0.6 and showing that each of its conditions holds:

- $\mathsf{G}_{\text{restricted}} \subseteq \mathsf{G}$ by definition of the GenMC-XMM restriction step.
- $\mathsf{G}_{\text{restricted}} \subseteq \mathsf{G}'$ by definition of the GenMC-XMM restriction step.

- f is injective, because only one write is matched to a $R_\perp$, so no two committed events in G can correspond to the same committed event in G'.
- $\forall c \in \mathcal{C}.\ \mathsf{f}(c) \neq \perp$ because if there are left-over $R_\perp$ at the end of execution, GenMC-XMM discards the graph.
- $\forall c \in \mathcal{C}.\ \mathsf{G}.\mathsf{tid}(\mathsf{f}(c)) = \mathsf{G}'.\mathsf{tid}(c)$ because either $c \in \mathsf{G}_{\text{restricted}}$, or c is a write matched to a $R_\perp$ that read from a write in G in the same thread as c.
- $\forall c \in \mathcal{C}.\ \mathsf{G}.\mathsf{lab}(\mathsf{f}(c)) = \mathsf{G}'.\mathsf{lab}(c)$ because either $c \in \mathsf{G}_{\text{restricted}}$, or c is a write matched to a $R_\perp$ that read from a write in G with the same label.
- $\mathsf{f} \uparrow ([\mathcal{C}]; \mathsf{G}'.\mathtt{rpo}; [\mathcal{C}]) = [\mathsf{f} \uparrow \mathcal{C}]; \mathsf{G}.\mathtt{rpo}; [\mathsf{f} \uparrow \mathcal{C}]$ because we have added a specific check (Section 6.9) that discards executions that do not satisfy this condition.
- $\mathsf{f} \uparrow ([\mathcal{C}]; \mathsf{G}'.\mathtt{rf}; [\mathcal{C}]) = [\mathsf{f} \uparrow \mathcal{C}]; \mathsf{G}.\mathtt{rf}; [\mathsf{f} \uparrow \mathcal{C}]$ because $\mathsf{G}_{\text{restricted}} \subseteq \mathsf{G}'$ so all rf edges in $\mathsf{G}_{\text{restricted}}$ remain the same. Committed writes matched to a $\mathrm{R}_\perp$ in G' have an equivalent write in G, so their rf edges are also the same.
- $\mathsf{f} \uparrow ([\mathcal{C}]; \mathsf{G}'.\mathtt{mo}; [\mathcal{C}]) = [\mathsf{f} \uparrow \mathcal{C}]; \mathsf{G}.\mathtt{mo}; [\mathsf{f} \uparrow \mathcal{C}]$ because we have added a specific check (Section 6.9) that discards executions that do not satisfy this condition.

□

Theorem 7.

$$\frac{\mathsf{G} \xrightarrow[\texttt{GenMC-XMM}]{(f,C)} \mathsf{G}' \qquad \mathcal{C} = \mathcal{C}_{\text{GenMC-XMM}}}{\mathit{StableUncommitedReads}(\mathsf{G}', \mathcal{C})}$$

Proof.

We have implemented the *stable uncommitted reads* Definition 5.0.7 using the pseudocode shown in Algorithm 5. The function *areUncommittedReadsStable* is called every time an execution is completed. If it returns false, the execution is discarded. □

Theorem 8.

$$\frac{\mathsf{G} \xrightarrow[\texttt{GenMC-XMM}]{(f,C)} \mathsf{G}'}{\mathit{Consistent}(\mathsf{G}')}$$

Proof.

We have implemented the consistency check from Definition 5.0.4, using the pseudocode shown in Algorithm 3. The *atomicity* and *SC* constraints are not checked directly by GenMC-XMM because GenMC already ensures that they are respected when it constructs the graphs. □

6.11 Limitations: Completeness and Optimality

GenMC-XMM is limited in the following aspects:

- GenMC-XMM is not a complete model checker because it approximates the set of committed events. A complete model checker would have to try all subsets of committed events, which would not scale since it requires $O(2^n)$ time complexity. A possible solution to this problem would be to prove that only certain subsets of committed events lead to XMM consistent executions, but so far, we have not found this true. In our tests, available in the artifact (Appendix A), we found that GenMC-XMM finds all XMM consistent executions in 73 tests, and misses an execution in 2: *LB+coh-cyc* and *LB+porf-suffix*.
- In the LB+coh-cyc test, GenMC-XMM misses an XMM consistent execution, shown in Figure 6.2 on the right. In order to obtain this execution, one should choose $C = \{\mathsf{W}(Y,1), \mathsf{R}(Y,1), \mathsf{W}(X,3)\}$ (highlighted in blue in the figure) and $\mathcal{D} = \emptyset$. Doing so allows $\mathsf{W}(X,1)$ in thread 2 to be added to the graph before $\mathsf{W}(X,2)$ in thread 1, which results in a change in mo order.

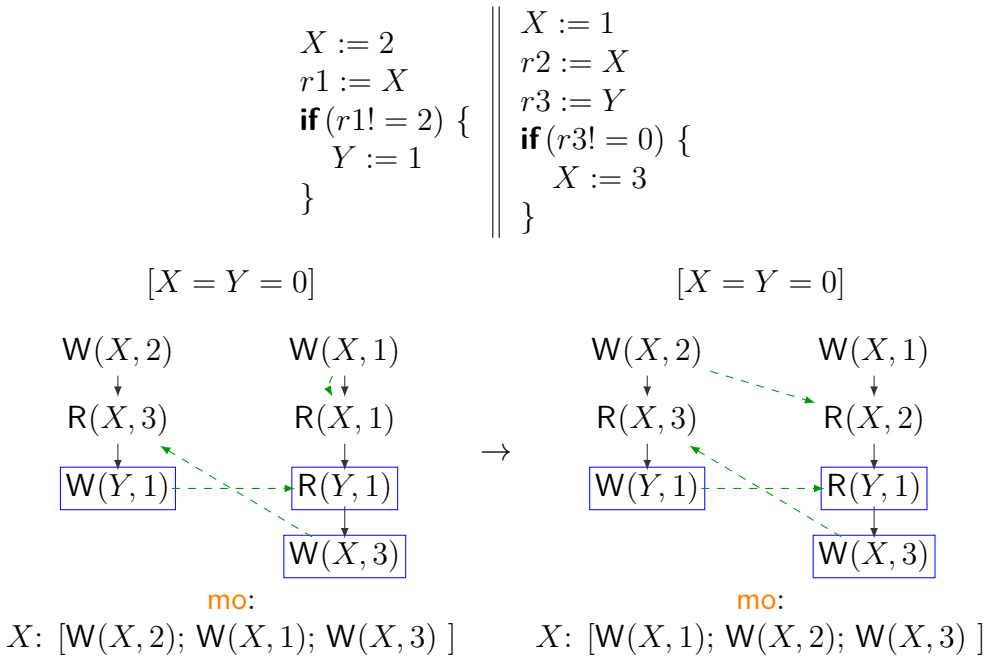


Figure 6.2: *LB+coh-cyc* test: GenMC-XMM fails to produce the graph on the right from the one on the left.

- Also, in the LB+porf-suffix test, which we have designed, GenMC-XMM is unable to produce the XMM consistent execution shown in Figure 6.3 on the right. The XMM semantics allow the graph on the right to be derived from the one on the left by selecting the committed events, highlighted in blue in the figure below, as $C = \{\mathsf{W}(Y,1), \mathsf{W}(Z,1), \mathsf{R}(Y,1), \mathsf{W}(X,1)\}$. However, GenMC-XMM over-approximates C, and selects all events highlighted in purple below: $C = \{\mathsf{W}(A,1), \mathsf{W}(Y,1), \mathsf{R}(A,1), \mathsf{W}(Z,1), \mathsf{R}(Y,1), \mathsf{W}(X,1)\}$. Because GenMC-XMM also selects $\mathsf{R}(A,1)$, the execution on the right cannot be derived. $\mathsf{R}(A,1)$ should change to $\mathsf{R}(B,1)$, and therefore should not be committed.
- GenMC prevents duplicates by marking backward revisits as "visited". This strategy enables GenMC to determine whether a revisit would lead to a duplicate graph before the re-execution step. GenMC-XMM cannot adopt this same strategy because different Load Buffering races can lead to the same execution (Section 6.7). Instead, it uses a hash set to record visited graphs. This strategy is less efficient because duplicate graphs must be fully built before being discarded, and it adds the overhead of having to hash the graphs.

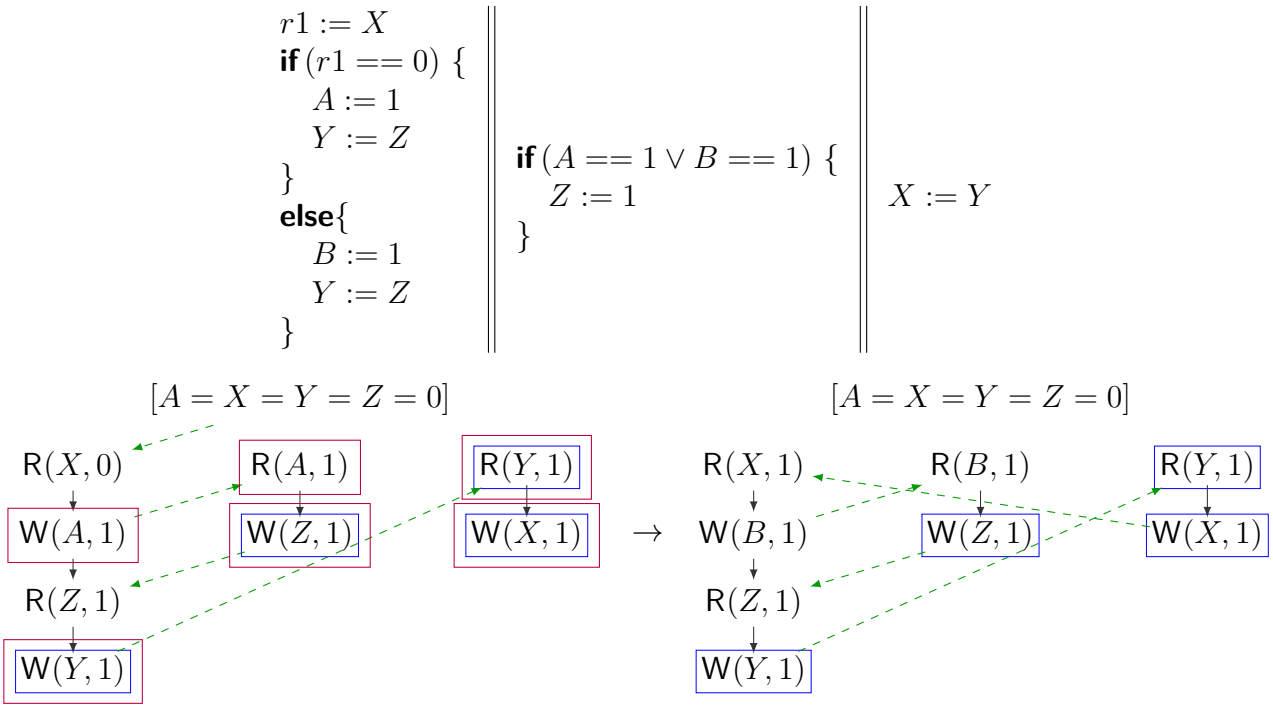


Figure 6.3: *LB+porf-suffix* test: GenMC-XMM fails to produce the graph on the right from the one on the left.

- GenMC checks the coherence property every time an event is added to a graph. GenMC-XMM cannot do the same because some reads may not have an rf edge. Instead, GenMC-XMM performs the check only once a graph is complete. GenMC's strategy is more efficient since it allows discarding incoherent graphs earlier.

Chapter 7

Evaluation

In this chapter, we evaluate the GenMC-XMM algorithm by comparing it to GenMC$_{\mathsf{RC11}}$, WMC, and GenMC$_{\mathsf{IMM}}$.

We are answering the following questions:

RQ1 How often does GenMC-XMM miss some XMM consistent executions? (Section 7.1)

RQ2 How is XMM different from Weakestmo2 in litmus tests? (Section 7.2)

RQ3 How effective is GenMC-XMM in verifying real-world lock-free data structures? (Section 7.3)

RQ4 How does GenMC-XMM scale in synthetic benchmarks compared to similar tools? (Section 7.4)

RQ5 How many duplicates does GenMC-XMM explore in load buffering tests? (Section 7.5)

The tools we are using for comparison with GenMC-XMM are:

- **GenMC$_{\mathsf{RC11}}$** [21] (Chapter 3) is the stateless model checker that GenMC-XMM is based upon.
- **Old GenMC$_{\mathsf{RC11}}$** is the old version of GenMC upon which WMC is based. We included this version in the data-structure benchmarks because we found some differences in running time and outputted executions compared to the current version.
- **WMC** [28] (Chapter 4) is a multi-execution stateless model checker based on Old GenMC$_{\mathsf{RC11}}$. It model checks the *Weakestmo2* multi-execution memory model [9].
- **GenMC$_{\mathsf{IMM}}$** [22] is a model checker based on GenMC, for the IMM (Intermediate Memory Model) [33]. It allows some cyclic executions by tracking dependencies. IMM is an abstraction over hardware memory models and allows language memory models such as RC11 and Promising Semantics to prove compilation correctness to IMM instead of to each hardware model.

The tests are well-known tests used in this field of research. They are taken from the benchmark suites of GenMC [21], WMC [28], Nidhugg [3], and RCMC [23]. We explain each test/benchmark we used in Appendix B.

The benchmarks were run in a Docker container on a macOS machine with a 2.9 GHz Intel Core i9 CPU and 32 GB of memory. The container is available for download by following

the instructions in Appendix A. The execution times reported in Tables 7.3, 7.5 and 7.7 are averaged over five runs.

7.1 Testing the completeness of GenMC-XMM [RQ1]

Unfortunately, it is tough to assess the completeness of an algorithm. We cannot provide mathematical guarantees about how many executions GenMC-XMM misses. The next best approach is to select a set of tests and manually verify whether GenMC-XMM finds all XMM consistent executions.

We have selected 73 tests, all containing load buffering races and relaxed atomics, and we found that GenMC-XMM missed an XMM consistent execution only in two cases: *LB+coh-cyc* and *LB+porf-suffix*. We explain why GenMC-XMM fails those tests in Section 6.11.

All tests, together with the manually computed XMM consistent executions, are available in the Docker container with the other benchmarks.

7.2 Comparing GenMC-XMM and WMC on litmus tests [RQ2]

Litmus tests are small, carefully crafted programs designed to expose and understand the behavior of concurrent systems under weak memory models.

How does the XMM model differ from Weakestmo2? To answer this question, we have compared the output of GenMC-XMM and WMC on 73 litmus tests with LB races taken from the GenMC-XMM benchmark suite. We found that in 12 of them, the two model checkers explored a non-equal number of executions (Table 7.1). In addition to the executions found by WMC and GenMC-XMM, we added the ones found by $\text{GenMC}_{\text{RC11}}$ and $\text{GenMC}_{\text{IMM}}$ in the two leftmost columns.

XMM finds more executions than the other model checkers in all 12 tests. RC11 and IMM have the same number of executions in all cases. RC11 and Weakestmo2 differ only in 5 tests: *R-bot-multi-matching*, *java-test19*, *java-test20*, *LB+seq-src*, and *java-test5*.

The test where Weakestmo2 and XMM diverge the most is *LB+coh+RR+cf*. In this test, RC11, IMM, and Weakestmo2 allow 24 executions, but XMM allows 36. This is because Weakestmo2 enforces a global event-structure level mo relation, whereas XMM does not have this constraint.

	Number of Executions			
Test Name	$\text{GenMC}_{\text{RC11}}$	$\text{GenMC}_{\text{IMM}}$	WMC	GenMC-XMM
LB+coh-cyc	5	5	5	6
LB+equals	9	9	9	10
LB-invis-write+dep	2	2	2	3
R-bot-multi-matching	15	18	20	21
java-test9a	10	10	10	12
LB+coh-cyc+Wd	10	10	10	12
java-test10	5	5	5	8
java-test19	14	14	17	20
java-test20	14	14	17	20
LB+seq-src	14	14	17	20
java-test5	20	20	24	28
LB+coh+RR+cf	24	24	24	36

Table 7.1: Number of executions found by $\text{GenMC}_{\text{RC11}}$, $\text{GenMC}_{\text{IMM}}$, WMC, and GenMC-XMM on litmus tests where XMM has different consistent executions than Weakestmo2

7.3 Evaluating GenMC-XMM on data-structure benchmarks [RQ3]

In this section, we compare GenMC-XMM to $\text{GenMC}_{\text{RC11}}$ and WMC on real-world lock-free data structures. Since WMC is based on an old version of GenMC, we also included this version under the name "Old $\text{GenMC}_{\text{RC11}}$". This older version does not support the *symmetry reduction optimization* [21], so we disabled it for $\text{GenMC}_{\text{RC11}}$, $\text{GenMC}_{\text{IMM}}$, and GenMC-XMM to have a fairer comparison.

In Table 7.2, we reported the number of executions explored by each model checker. We highlighted in blue the tests where GenMC-XMM's output was different from $\text{GenMC}_{\text{RC11}}$, and in red the ones where the results of the model checkers differed. In Table 7.3, we reported the time it took to verify each test. The last column of each table is called "LB races" and represents the number of Load Buffering races explored by GenMC-XMM. The timeout was set to 60s.

All selected tests have load buffering races, but GenMC-XMM finds more executions than $\text{GenMC}_{\text{RC11}}$ only in two tests: *chase-lev* and *dq*. In *chase-lev*, $\text{GenMC}_{\text{RC11}}$ finds 3639 executions, and GenMC-XMM 3821. In *dq* $\text{GenMC}_{\text{RC11}}$ finds 1924, and GenMC-XMM 2065.

In all tests, GenMC-XMM finds more, or the same, executions compared to the other tools. We expected this, because XMM is in general a weaker model compared to RC11, IMM, and Weakestmo2.

In *chase-lev* and *dq*, Old $\text{GenMC}_{\text{RC11}}$ reports different executions than $\text{GenMC}_{\text{RC11}}$. This is due to bug fixes and improvements added to $\text{GenMC}_{\text{RC11}}$ after WMC was released. Old GenMC does not complete *linuxrwlocks* and *fcombiner-async* within the timeout, and consequently neither does WMC.

WMC reports the same executions as Old $\text{GenMC}_{\text{RC11}}$, except for *chase-lev* where Old $\text{GenMC}_{\text{RC11}}$ outputs 3809 executions, and WMC 3975.

$\text{GenMC}_{\text{IMM}}$ reports the same executions as $\text{GenMC}_{\text{RC11}}$.

As shown in Figure 7.1, in all tests with many load buffering races, GenMC-XMM performs worse than GenMC$_{\mathsf{RC11}}$. The tests where the performance of GenMC-XMM performs worst are *mpmc-queue-bnd* and *ticketlock*, the tests with the most LB races. A plausible reason is that each load buffering race leads to a new execution to explore, even if it may be discarded. Additionally, GenMC-XMM is designed to create a copy of the current graph whenever an LB race is encountered, which is an expensive operation. In tests with few LB races, the performance of the two algorithms is comparable.

WMC performs surprisingly well in all tests and does not appear to be affected by LB races as much as GenMC-XMM. It performs worse than GenMC$_{\mathsf{RC11}}$ only when Old GenMC$_{\mathsf{RC11}}$ also performs worse.

	Number of Executions					
Test Name	Old GenMC$_{\mathsf{RC11}}$	WMC	GenMC$_{\mathsf{RC11}}$	GenMC$_{\mathsf{IMM}}$	GenMC-XMM	LB Races
mpmc-queue-bnd	15752	15752	15752	15752	15752	24240
ticketlock	720	720	720	720	720	10800
chase-lev	3809	3975	3639	3639	3821	6530
buf-ring	1218	1218	1218	1218	1218	2172
dq	1802	1802	1924	1924	2065	1616
stc	183	183	183	183	183	1515
linuxrwlocks	⏳	⏳	216	216	216	384
twalock	96	96	96	96	96	288
fcombiner-async	⏳	⏳	24	24	24	32
mutex	12	12	12	12	12	30

Table 7.2: Number of executions explored on data structure benchmarks. GenMC-XMM explores more executions than other tools in the *chase-lev* and *dq* tests.

	Execution Time					
Test Name	Old GenMC$_{\mathsf{RC11}}$	WMC	GenMC$_{\mathsf{RC11}}$	GenMC$_{\mathsf{IMM}}$	GenMC-XMM	LB Races
mpmc-queue-bnd	4.00s	4.31s	4.81s	9.26s	12.64s	24240
ticketlock	3.40s	3.65s	0.34s	2.01s	48.43s	10800
chase-lev	0.58s	0.66s	0.74s	1.91s	1.95s	6530
buf-ring	1.54s	1.59s	1.72s	3.92s	2.28s	2172
dq	0.14s	0.15s	0.18s	0.28s	2.79s	1616
stc	0.05s	0.07s	0.14s	0.15s	0.30s	1515
linuxrwlocks	⏳	⏳	0.23s	0.40s	0.81s	384
twalock	0.03s	0.03s	0.34s	0.34s	0.41s	288
fcombiner-async	⏳	⏳	0.33s	0.36s	0.34s	32
mutex	0.02s	0.02s	0.03s	0.05s	0.04s	30

Table 7.3: Execution time on data structure benchmarks. GenMC-XMM's execution time is comparable to that of the other tools, except in the *ticketlock* and *mpmc-queue-bnd* tests where it takes longer.

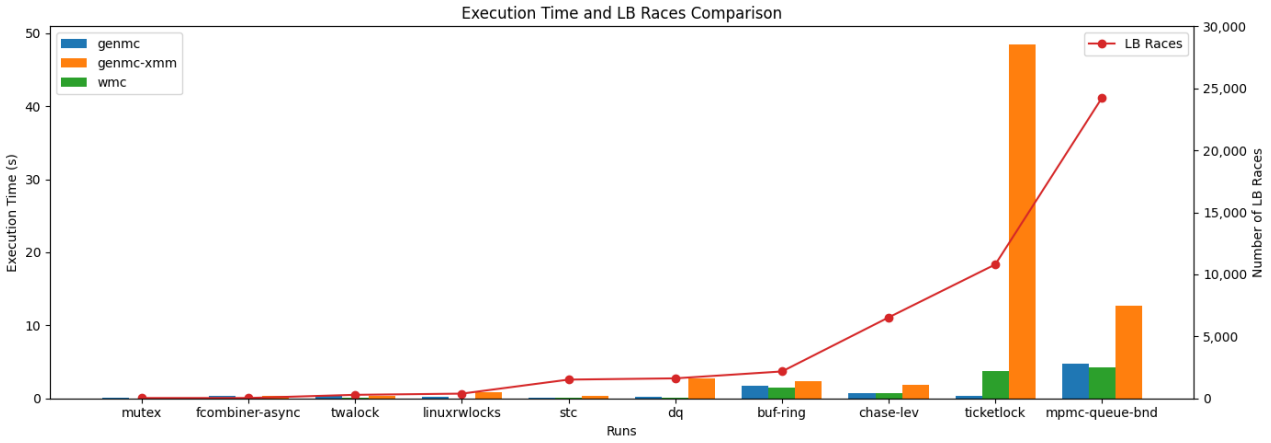


Figure 7.1: Execution times and LB races on data structure tests. GenMC-XMM's execution time is comparable to other tools, except when the number of LB races exceeds 10,000.

7.4 Evaluating GenMC-XMM on synthetic benchmarks [RQ4]

In this section, we compare how GenMC-XMM, "Old GenMC$_{\text{RC11}}$", WMC, and GenMC$_{\text{RC11}}$ scale on synthetic benchmarks designed to have thousands of executions to explore (tables 7.4 and 7.5). In the last column on the right, we report the number of Load Buffering races explored by GenMC-XMM. The rows where GenMC-XMM's output differed from GenMC$_{\text{RC11}}$ are highlighted in blue. The rows where any model checker outputs a different result are highlighted in red. Some tests have a number in parentheses next to their name that indicates how many threads were created in that test. The timeout was set to 60s.

Table 7.4 shows the number of executions explored by each tool. We make the following considerations about the number of executions explored by the tools:

- GenMC$_{\text{RC11}}$and Old GenMC$_{\text{RC11}}$explored different executions in tests *szymanski*, and *dekker-bnd*. This is due to fixes and improvements introduced after WMC was released.
- WMC explored the same executions as *Old GenMC*$_{\text{RC11}}$ in all tests, except in *szymanski*.
- GenMC-XMM also finds cyclic executions only in *szymanski*. In *szymanski(1)* and *szymanski(2)*, Old GenMC$_{\text{RC11}}$finds more executions than GenMC$_{\text{RC11}}$. This causes WMC also to have more total executions than GenMC-XMM. However, if we count only the cyclic executions for *szymanski(1)*, we get that WMC found $44-40=4$ cyclic executions, and GenMC-XMM found $36-32=4$. For *szymanski(2)*, WMC found $6344-5738=606$ cyclic executions, and GenMC-XMM found $2930-2216=714$.

Table 7.5 shows the execution time of each tool. We make the following considerations about the execution time of the tools:

- In all tests without LB races, GenMC-XMM has a slight performance overhead compared to GenMC$_{\text{RC11}}$because GenMC-XMM has to look for LB races while GenMC$_{\text{RC11}}$does not. This overhead is so small that it was not accurately measured in the execution times.
- In all tests with LB races and a parameter that defines the number of threads created (*szymanski*, *ainc*, *casrot*, *casw*), the number of LB races encountered grows with the number of threads created. This significantly impacts the execution time of GenMC-XMM: while GenMC and WMC finish most tests in under a second, GenMC-XMM

finishes in tens of seconds. We have plotted the number of LB races and execution times of GenMC-XMM and GenMC in Figure 7.2.

These tests show that GenMC-XMM does not scale as well as GenMC and WMC as the number of load buffering races increases. After investigating the algorithm with a CPU profiler, we found that this is caused by GenMC-XMM making a copy of the current graph each time an LB race is encountered. This copy is needed for the current algorithm design to work, but it is possible that the design could be modified to avoid copying the graph. However, we do not expect many real-world scenarios to have this many load buffering races, so we leave this performance improvement for future work. WMC appears to be much less affected by the number of LB races than GenMC-XMM. After inspection of the WMC code, we found that WMC does not copy the current graph every time an LB race is found, which would explain its better scalability compared to GenMC-XMM.

	Number of Executions					
Test Name	Old GenMC$_{\mathsf{RC11}}$	WMC	GenMC$_{\mathsf{RC11}}$	GenMC$_{\mathsf{IMM}}$	GenMC-XMM	LB Races
szymanski(1)	40	44	32	32	36	40
szymanski(2)	5738	6344	2216	2216	2930	15176
dekker-bnd	59	59	55	55	55	123
reorder2	1296	1296	1296	1296	1296	0
fib-bench	34205	34205	34205	34205	34205	0
indexer(14)	512	512	512	512	512	0
indexer(15)	4096	4096	4096	4096	4096	0
ainc(6)	720	720	720	720	720	10800
ainc(7)	5040	5040	5040	5040	5040	105840
casrot(9)	8597	8597	8597	8597	8597	42320
casrot(10)	38486	38486	38486	38486	38486	223701
casw(4)	1200	1200	1200	1200	1200	4800
casw(5)	32880	32880	32880	32880	32880	192480

Table 7.4: Number of executions explored on synthetic benchmarks. GenMC-XMM finds the same number of executions as other tools except in the *szymanski* test.

	Execution Time					
Test Name	Old GenMC$_{\mathsf{RC11}}$	WMC	GenMC$_{\mathsf{RC11}}$	GenMC$_{\mathsf{IMM}}$	GenMC-XMM	LB Races
szymanski(1)	0.02s	0.00s	0.04s	0.04s	0.04s	40
szymanski(2)	0.30s	0.36s	0.23s	0.47s	2.03s	15176
dekker-bnd	0.01s	0.01s	0.04s	0.04s	0.04s	123
reorder2	0.11s	0.17s	0.13s	0.18s	0.12s	0
fib-bench	1.21s	1.12s	0.85s	0.86s	0.91s	0
indexer(14)	0.33s	0.35s	0.63s	1.69s	0.69s	0
indexer(15)	2.90s	2.89s	5.44s	13.11s	5.20s	0
ainc(6)	0.03s	0.04s	0.07s	0.10s	1.24s	10800
ainc(7)	0.25s	0.36s	0.34s	0.59s	14.59s	105840
casrot(9)	0.44s	0.47s	0.23s	0.35s	6.87s	42320
casrot(10)	2.34s	2.20s	0.95s	1.51s	42.11s	223701
casw(4)	0.06s	0.06s	0.06s	0.10s	0.34s	4800
casw(5)	1.62s	1.84s	0.76s	1.27s	13.29s	192480

Table 7.5: Execution times on synthetic benchmarks. GenMC-XMM does not scale well on tests with many LB races and can take much longer than other tools.

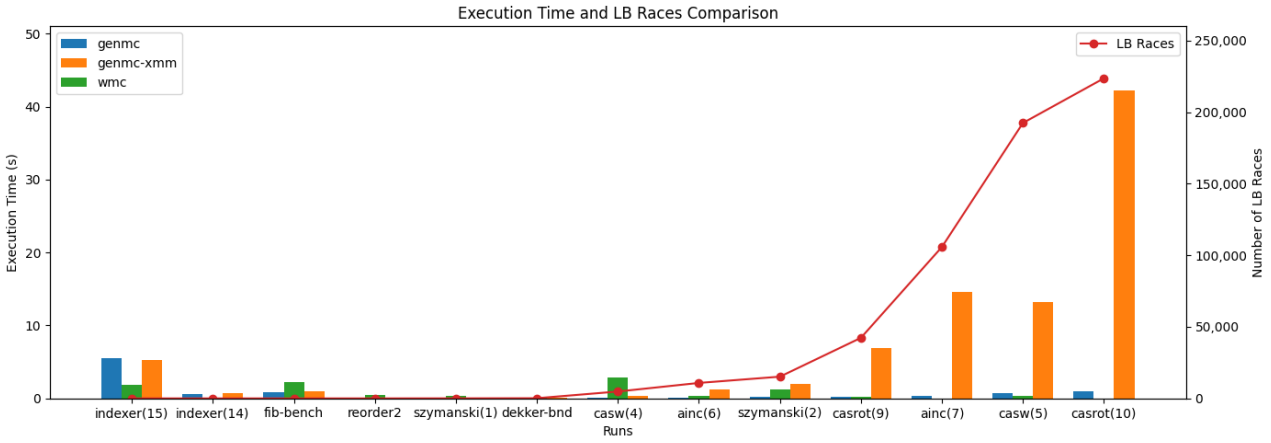


Figure 7.2: Execution times and LB races on synthetic tests. As the number of LB races increases, so does the difference in execution time between GenMC-XMM and other tools.

7.5 Evaluating the number of duplicate graphs explored by GenMC-XMM [RQ5]

In this section, we compare GenMC-XMM, WMC, and GenMC$_{\mathsf{RC11}}$ on a set of artificial tests designed to contain load buffering races, taken from [28]. In Table 7.6, we report the number of executions explored by GenMC$_{\mathsf{RC11}}$, GenMC$_{\mathsf{IMM}}$, WMC, and GenMC-XMM, the number of duplicates explored by WMC and GenMC-XMM, and the number of load buffering races found during exploration by GenMC-XMM. In Table 7.7, we report the execution time of each model checker.

In *LBn+ctrl*, *LBn+data*, and *LBn*, GenMC$_{\mathsf{IMM}}$, WMC, and GenMC-XMM report only one more execution than GenMC$_{\mathsf{RC11}}$. The number of duplicates explored in these tests stays low, with GenMC-XMM reporting one more than WMC for all these test cases.

LBn-pairs consists of n threads, divided into pairs of (writer, reader), writing/reading to/from pairs of variables. This causes 4^n, where n is the number of pairs, executions to be allowed by GenMC$_{\mathsf{IMM}}$, WMC, and GenMC-XMM. Neither WMC nor GenMC-XMM scale well due to this test's exponential number of outcomes, with the number of duplicate executions growing faster than the number of consistent executions. In *LBn-pairs(14)*, WMC explores 61741 duplicates and GenMC-XMM 48250.

GenMC$_{\mathsf{RC11}}$ is the faster model checker overall. GenMC$_{\mathsf{IMM}}$, WMC, and GenMC-XMM are comparable in terms of performance, except for the *LBn-pairs* test, where GenMC-XMM is considerably slower than the other tools. The performance penalty of GenMC-XMM in this test is due to the high number of LB races, which causes many graph copies to be created like in the synthetic tests of Section 7.4.

	Number of Executions				Number of Duplicates		
Test Name	$\text{GenMC}_{\text{RC11}}$	$\text{GenMC}_{\text{IMM}}$	WMC	GenMC-XMM	WMC	GenMC-XMM	LB Races
LBn+ctrl(10)	9	11	11	11	0	1	1
LBn+ctrl(12)	10	11	11	11	0	1	1
LBn+ctrl(14)	10	11	11	11	0	1	1
LBn+data(10)	19	1024	1024	1024	0	1	10
LBn+data(12)	1023	1024	1024	1024	0	1	10
LBn+data(14)	1023	1024	1024	1024	0	1	10
LBn(10)	1023	1024	1024	1024	9	10	10
LBn(12)	4095	4096	4096	4096	11	12	12
LBn(14)	16383	16384	16384	16384	13	14	14
LBn-pairs(10)	243	1024	1024	1024	2101	2184	2560
LBn-pairs(12)	729	4096	4096	4096	11529	10379	12288
LBn-pairs(14)	2187	16384	16384	16384	61741	48250	57344

Table 7.6: Load Buffering benchmarks. GenMC-XMM finds the same number of cyclic executions as other tools. The number of duplicates visited is slightly less than that of WMC.

	Execution Time				
Test Name	$\text{GenMC}_{\text{RC11}}$	$\text{GenMC}_{\text{IMM}}$	WMC	GenMC-XMM	LB Races
LBn+ctrl(10)	0.06s	0.05s	0.01s	0.04s	1
LBn+ctrl(12)	0.05s	0.04s	0.01s	0.04s	1
LBn+ctrl(14)	0.04s	0.04s	0.01s	0.05s	1
LBn+data(10)	0.04s	0.19s	0.10s	0.15s	10
LBn+data(12)	0.10s	0.18s	0.10s	0.13s	10
LBn+data(14)	0.11s	0.17s	0.10s	0.11s	10
LBn(10)	0.11s	0.14s	0.12s	0.13s	10
LBn(12)	0.33s	0.55s	0.46s	0.40s	12
LBn(14)	1.37s	2.40s	1.68s	1.47s	14
LBn-pairs(10)	0.06s	0.19s	0.34s	1.43s	2560
LBn-pairs(12)	0.12s	0.65s	2.27s	7.06s	12288
LBn-pairs(14)	0.34s	3.19s	11.60s	38.12s	57344

Table 7.7: Load Buffering time benchmarks. GenMC-XMM takes the same time as other tools when there are few LB races. It takes longer when the number of LB races increases.

Chapter 8

Related Work

This work builds on top of a series of attempts at designing a language memory model and accompanying model checker that does not suffer from the "out-of-thin-air" (OOTA) problem [8] and allows for the most efficient compilation on all computer architectures. This has been the "holy grail" of weak memory consistency, and after decades of work by researchers, a perfect solution still does not exist.

8.1 Language Memory Models

The first mainstream language to receive a memory model was Java [27] back in 1996, and then a revised version in 2004. This model was criticized for not supporting certain desirable program transformations [35, 38].

On the contrary, the original C++ memory model [6] was criticized because, for the sake of supporting more program transformations, it allowed OOTA executions. These are undesirable because they render logical reasoning about programs ineffective [40]. Consider the following *LB+deps* example:

$$\begin{array}{l|l|l} \text{if (X == 1)} & \| & \text{if (Y == 1)} \\ Y := 1; & \| & X := 1; \end{array}$$

Because of the dependencies created by the *if* statements, the only possible outcome of this program is $X = Y = 0$. Nonetheless, the original C11 model allows the OOTA outcome $X = Y = 1$, invalidating the thread-local reasoning of constant propagation [42].

A new model called *Repaired C11*, or RC11 [24], was proposed to patch this issue. This model aimed to strengthen the C11 model enough to exclude OOTA outcomes, even if this entailed slightly less efficient compilation. Part of the solution proposed by the authors is to require po∪rf to be acyclic. This has the undesirable effect of also ruling out the following *LB* behavior:

$$\begin{array}{l|l|l} a := X & \| & b := Y \\ Y := 1 & \| & X := b \end{array}$$

On Arm and Power architectures, the outcome $a = b = 1$ is possible, but not according to *RC11*. To make Arm and Power comply with *RC11*, the compilation to these architectures must add fences to prevent reordering of instructions. The performance overhead of this less

efficient compilation has been estimated in [31] and was found to be as high as 17% in some benchmarks, but on average, about 3%.

The Java memory model was updated [7] when Java was extended with different memory access modes. This new model adopts a per-execution style, and like RC11, it forbids po∪rf cycles.

Since then, researchers have strived to design a weak memory model that did not exhibit OOTA executions and enabled as many compiler optimizations as possible by allowing weak behaviors. These new advanced models had to adopt a multi-execution reasoning style to achieve this goal. With this paradigm, multiple executions are considered together to justify a program outcome [28]. Notable examples are:

- **Promising Semantics** [19], achieves efficient compilation by allowing threads to *promise* a write and other threads to read from either a normal write or a promise. Each promise must be *certified*, i.e., show that the thread that made the promise can perform the write it promised. In a later paper, this model was updated to support transformations based on global analysis [26]. On the downside, both versions suffered from a high degree of complexity, which may be why no model checking algorithm for the entire model has been designed.
- **Weakestmo** [9], represents multiple executions together explicitly in a event structure graph [43]. In other words, a single event structure graph can contain multiple execution branches of the program. The original model was presented without a model checker. Still, a slightly strengthened Weakestmo2 model was proposed years later [28] with an automatic checker tool called WMC (Chapter 4).

The downside of multi-execution models is that they are significantly more complicated than their single-execution counterparts. Researchers are now trying to design simpler models that achieve the same or better results. We believe the XMM memory model [34] and our GenMC-XMM model checker tool are yet another step towards this goal.

8.2 Model Checkers

From the perspective of program verification, much work has been put into developing model checker tools for porf-acyclic models (e.g. [1, 2, 4, 5, 11, 14, 15, 17, 18, 20, 23, 44]). GenMC [21], which forms the basis of GenMC-XMM, belongs to this category. This state-of-the-art tool is a stateless [15] model checker, meaning that it explores the states of a program without recording the past ones. It does so by keeping track of the current execution and deriving the next consistent execution directly from it.

Compared to acyclic model checking, porf-cyclic models have been explored less. This could be because this class of models allows weak outcomes that result in extra complexity for the model checker. One of the first porf-cyclic models was designed in 2013 and is called CDSChecker [30]. This tool was designed for the original C11 memory model, allowing loads to read from future stores with a system of promises similar to *Promising Semantics* [19]. Another notable tool is WMC [28], which targets the Weakestmo2 memory model. Like our solution, it is based on GenMC [21], and it makes use of the *Load Buffering Race Freedom* theorem [28] to determine when to explore cyclic executions. A limitation of Weakestmo2 compared to XMM is that it does not support the sequentialization transformation.

Chapter 9

Conclusion

XMM [34] is a new memory model designed to rule out OOTA behaviors while supporting common compiler optimizations such as reorderings and sequentialization. Compared to the state-of-the-art multi-execution relaxed memory models, such as Weakestmo [9] and Promising [19], XMM is considerably simpler and supports the sequentialization optimization. Weakestmo is based on Event Structure Theory and uses Event Structures to represent multiple program branches together. Promising uses a concept of thread promises and certification to enable reading from speculative writes. XMM uses the concept of committing events and graph re-execution. The Promising model lacks a complete model checker. Weakestmo has a model checker called WMC [28], designed for a strengthened version of the model called Weakestmo2.

In this work, we have presented a model checker for the XMM memory model called GenMC-XMM. Our algorithm is based on GenMC, a stateless model checker for RC11 [24] and IMM [33]. It uses XMM's *Load Buffering Race Freedom* property, which states that in the absence of Load Buffering Races, XMM behaves like RC11. When a Load Buffering race is present in an execution graph, GenMC-XMM restricts the graph and re-executes it, obtaining a cyclic execution. A cyclic graph can be further restricted to obtain new executions. With this simple design based on 1. searching for Load Buffering races, 2. restricting the graph, 3. re-executing, and 4. repeating, GenMC-XMM outputted all XMM executions on 71 litmus tests with load buffering races. However, in two cases, GenMC-XMM did not enumerate all of them: *LB+coh-cyc* and *LB+porf-prefix*.

GenMC-XMM is not a complete algorithm. Due to an algorithmic time complexity issue, a complete model checker tool for XMM would have required $O(2^n)$ where n is the size of a graph. However, we have proven that it is sound. We do not know of any other model checker algorithm for a multi-execution memory model that has been proven sound until now.

Our evaluation determined that GenMC-XMM takes more time to verify a program when it exhibits upwards of thousands of Load Buffering data races, which should be rare. We believe that GenMC-XMM's performance could be improved by avoiding copying the current graph at each LB race, discarding duplicates before re-executing, and checking for consistency during re-execution.

XMM and GenMC-XMM are a significant step towards simpler multi-execution memory models that allow more compiler optimizations and no unnecessary fences when compiling to weak architectures.

Appendices

Appendix A

Evaluation Reproduction Instructions

We provide a Docker image with the GenMC-XMM code, and all the necessary tools to reproduce our results. The instructions to set this up are the following:

1. Download and install Docker and Docker Compose from `https://docs.docker.com/engine/install`.
2. Clone the GitHub repository with the Docker image from `https://github.com/matteo-meluzzi/xmm-benchmarks`.
3. In the cloned "xmm-benchmarks" folder, enter the following command:

```
$ docker compose up
```

4. Docker should now start building the image. Once it is finished, the benchmarks can be accessed via a web browser at `http://localhost:8888`.
5. Each *.ipynb* file corresponds to a different benchmark. The "run-tests.py" file can be run to verify the GenMC-XMM output against its test suite.

Appendix B

Benchmark descriptions

B.1 Litmus tests

- **LB+acq** is like the classic LB test, but the loads use acquire memory order.
- **LB+coh-cyc** is taken from [10]. In this example, the outcome $r1 = 3 \wedge r2 = 2 \wedge r3 = 1$ is allowed by Promising, and forbidden by Weakestmo and XMM.
- **LB+equals** is a test where a compiler could potentially find invariant $X = Y$, and remove the redundant if statement if $(X == Y)$.
- **LB+rel** is like the classic LB test, but the stores use release memory order.
- **LB-invis-write+dep** is a test designed to verify that the execution obtained after applying the sequentialization optimization is outputted.
- **R-bot-multi-matching** is a test designed to verify that all writes that are compatible with a certain R_bot are matched.
- **java-test9a** is equal to Java causality test 9, taken from [27], except that variable X is initialized to 2, and thread 3 writes 0 to X instead of 2.
- **LB+coh-cyc+Wd** is similar to *LB+coh-cyc*, but with a read from a distinct location in the second thread and a write to this location in a separate thread. $r1 = 3 \wedge r2 = 2 \wedge r3 = 1$ should be still forbidden.
- **java-test10** is Java causality test 10 [27]. The Java model forbids outcome $r1 = 1 \wedge r2 = 1 \wedge r3 = 0$, but it is allowed by XMM.
- **java-test19** is Java causality test 19 [27]. Both the Java model, and XMM allow outcome $r1 = 42 \wedge r2 = 42 \wedge r3 = 42$.
- **java-test20** is Java causality test 20 [27]. Both the Java model, and XMM allow outcome $r1 = 42 \wedge r2 = 42 \wedge r3 = 42$.
- **LB+seq** is a test taken from the Promising paper [19]. It tests whether the sequentialization optimization is allowed.
- **java-test5** is Java causality test 5 [27]. The Java model forbids outcome $r1 = 1 \wedge r2 = 1 \wedge r3 == 0$, but XMM allows it.
- **LB+coh+RR+cf** this test shows between the global mo order of the Weakestmo model, and the absence of such constraint in the XMM model. Due to the absence of this

constraint, XMM allows more executions than Weakestmo in this test.

B.2 Data Structure Benchmarks

- **mpmc-queue-bnd** is a multi-producer multi-consumer queue.
- **ticketlock** is an implementation of a ticketlock algorithm.
- **chase-lev** is a dynamic circular work-stealing deque [12].
- **buf-ring** is a multi-producer multi-consumer ring buffer.
- **dq** is a multi-producer multi-consumer deque.
- **stc** is a stack implementation using Treiber's algorithm.
- **linuxrwlocks** is a read-write lock ported from the Linux kernel [20].
- **twalock** is a ticket lock augmented with a waiting array [13].
- **fcombiner** implements concurrent access to a set data structure using the flat combiner access paradigm [16].
- **mutex** is a mutual exclusive lock.

B.3 Synthetic Benchmarks

- **reorder2**: 2 threads write to 2 variables, while 2 other threads read from them.
- **fib-bench**: 2 threads compute the Fibonacci sequence using two shared variables.
- **szymanski**: is an implementation of Szymański's Mutual Exclusion Algorithm [39].
- **dekker-bnd**: is an implementation of Dekker's Mutual Exclusion algorithm.
- **indexer**: N threads modify a shared array.
- **ainc**: N threads increment a shared variable by 1.
- **casrot**: N threads compare-and-swap a shared variable, rotating its value.
- **casw**: N threads compare-and-swap a shared variable, and write a new value to it.

B.4 Load Buffering Benchmarks

- **LBn+ctrl**: N threads with a load buffering pattern spanning through all threads. All threads except the first one have a real control dependency between the load and the store.
- **LBn+data**: N threads with a load buffering pattern spanning through all threads. All threads except the first one have a real data dependency between the load and the store.
- **LBn**: N threads with a load buffering pattern spanning through all threads.
- **LBn-pairs**: N threads divided in pairs. Each pair reads/writes from/to a unique pair of variables forming a load buffering race.

Bibliography

[1] Parosh Abdulla et al. "Optimal dynamic partial order reduction". In: *SIGPLAN Not.* 49.1 (Jan. 2014), pp. 373–384. DOI: 10.1145/2578855.2535845.

[2] Parosh Aziz Abdulla et al. "Optimal stateless model checking under the release-acquire semantics". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276505.

[3] Parosh Aziz Abdulla et al. "Stateless Model Checking for POWER". In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pp. 134–156. DOI: 10.1007/978-3-319-41540-6_8.

[4] Parosh Aziz Abdulla et al. "Stateless Model Checking for TSO and PSO". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 353–367. DOI: 10.48550/arXiv.1501.02069.

[5] Elvira Albert et al. "Constrained Dynamic Partial Order Reduction". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 392–410. DOI: 10.1007/978-3-319-96142-2_24.

[6] Mark Batty et al. "Mathematizing C++ concurrency". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 55–66. DOI: 10.1145/1926385.1926394.

[7] John Bender and Jens Palsberg. "A formalization of Java's concurrent access modes". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360568.

[8] Hans-J. Boehm and Brian Demsky. "Outlawing ghosts: avoiding out-of-thin-air results". In: *Proceedings of the Workshop on Memory Systems Performance and Correctness*. MSPC '14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014. DOI: 10.1145/2618128.2618134.

[9] Soham Chakraborty and Viktor Vafeiadis. "Grounding thin-air reads with event structures". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290383.

[10] Soham Chakraborty and Viktor Vafeiadis. "Grounding thin-air reads with event structures". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290383.

[11] Marek Chalupa et al. "Data-centric dynamic partial order reduction". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158119.

[12] David Chase and Yossi Lev. "Dynamic circular work-stealing deque". In: *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '05. Las Vegas, Nevada, USA: Association for Computing Machinery, 2005, pp. 21–28. DOI: 10.1145/1073970.1073974.

[13] Dave Dice and Alex Kogan. "TWA – Ticket Locks Augmented with a Waiting Array". In: *Euro-Par 2019: Parallel Processing*. Ed. by Ramin Yahyapour. Cham: Springer International Publishing, 2019, pp. 334–345. DOI: 10.1007/978-3-030-29400-7_24.

[14] Cormac Flanagan and Patrice Godefroid. "Dynamic partial-order reduction for model checking software". In: *SIGPLAN Not.* 40.1 (Jan. 2005), pp. 110–121. DOI: 10.1145/1047659.1040315.

[15] Patrice Godefroid. “Model checking for programming languages using VeriSoft”. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’97. Paris, France: Association for Computing Machinery, 1997, pp. 174–186. DOI: 10.1145/263699.263717.

[16] Danny Hendler et al. “Flat combining and the synchronization-parallelism tradeoff”. In: *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’10. Thira, Santorini, Greece: Association for Computing Machinery, 2010, pp. 355–364. DOI: 10.1145/1810479.1810540.

[17] Jeff Huang. “Stateless model checking concurrent programs with maximal causality reduction”. In: *SIGPLAN Not.* 50.6 (June 2015), pp. 165–174. DOI: 10.1145/2813885.2737975.

[18] Shiyou Huang and Jeff Huang. “Maximal causality reduction for TSO and PSO”. In: *SIGPLAN Not.* 51.10 (Oct. 2016), pp. 447–461. DOI: 10.1145/3022671.2984025.

[19] Jeehoon Kang et al. “A promising semantics for relaxed-memory concurrency”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’17. Paris, France: Association for Computing Machinery, 2017, pp. 175–189. DOI: 10.1145/3009837.3009850.

[20] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. “Model checking for weakly consistent libraries”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 96–110. DOI: 10.1145/3314221.3314609.

[21] Michalis Kokologiannakis and Viktor Vafeiadis. “GenMC: A Model Checker for Weak Memory Models”. In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 427–440. DOI: 10.1007/978-3-030-81685-8_20.

[22] Michalis Kokologiannakis and Viktor Vafeiadis. “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. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 1157–1171. DOI: 10.1145/3373376.3378480.

[23] Michalis Kokologiannakis et al. “Effective stateless model checking for C/C++ concurrency”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158105.

[24] Ori Lahav et al. “Repairing sequential consistency in C/C++11”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 618–632. DOI: 10.1145/3062341.3062352.

[25] Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439.

[26] Sung-Hwan Lee et al. “Promising 2.0: global optimizations in relaxed memory concurrency”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 362–376. DOI: 10.1145/3385412.3386010.

[27] Jeremy Manson, William Pugh, and Sarita V. Adve. “The Java memory model”. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’05. Long Beach, California, USA: Association for Computing Machinery, 2005, pp. 378–391. DOI: 10.1145/1040305.1040336.

[28] Evgenii Moiseenko, Michalis Kokologiannakis, and Viktor Vafeiadis. “Model checking for a multi-execution memory model”. In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022). DOI: 10.1145/3563315.

[29] Evgenii Moiseenko et al. “Reconciling Event Structures with Modern Multiprocessors”. In: *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Ed. by Robert Hirschfeld and Tobias Pape. Vol. 166. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, 5:1–5:26. DOI: 10.4230/LIPIcs.ECOOP.2020.5.

[30] Brian Norris and Brian Demsky. “CDSchecker: checking concurrent data structures written with C/C++ atomics”. In: *SIGPLAN Not.* 48.10 (Oct. 2013), pp. 131–150. DOI: 10.1145/2544173.2509514.

[31] Peizhao Ou and Brian Demsky. “Towards understanding the costs of avoiding out-of-thin-air results”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276506.

[32] Scott Owens, Susmit Sarkar, and Peter Sewell. “A Better x86 Memory Model: x86-TSO”. In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 391–407. DOI: 10.1007/978-3-642-03359-9_27.

[33] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. “Bridging the gap between programming languages and hardware weak memory models”. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290382.

[34] Anton Podkopaev et al. “JMM-Like Relaxed Memory Model”. In review. 2024.

[35] William Pugh. “Fixing the Java memory model”. In: *Proceedings of the ACM 1999 Conference on Java Grande*. JAVA ’99. San Francisco, California, USA: Association for Computing Machinery, 1999, pp. 89–98. DOI: 10.1145/304065.304106.

[36] Christopher Pulte et al. “Promising-ARM/RISC-V: a simpler and faster operational concurrency model”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1–15. DOI: 10.1145/3314221.3314624.

[37] Christopher Pulte et al. “Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158107.

[38] Jaroslav Ševčík and David Aspinall. “On Validity of Program Transformations in the Java Memory Model”. In: *ECOOP 2008 – Object-Oriented Programming*. Ed. by Jan Vitek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 27–51. DOI: 10.1007/978-3-540-70592-5_3.

[39] B. K. Szymanski. “A simple solution to Lamport’s concurrent programming problem with linear wait”. In: *Proceedings of the 2nd International Conference on Supercomputing*. ICS ’88. St. Malo, France: Association for Computing Machinery, 1988, pp. 621–626. DOI: 10.1145/55364.55425.

[40] Viktor Vafeiadis and Chinmay Narayan. “Relaxed separation logic: a program logic for C11 concurrency”. In: *SIGPLAN Not.* 48.10 (Oct. 2013), pp. 867–884. DOI: 10.1145/2544173.2509532.

[41] Viktor Vafeiadis et al. “Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it”. In: *SIGPLAN Not.* 50.1 (Jan. 2015), pp. 209–220. DOI: 10.1145/2775051.2676995.

[42] Mark N. Wegman and F. Kenneth Zadeck. “Constant propagation with conditional branches”. In: *ACM Trans. Program. Lang. Syst.* 13.2 (Apr. 1991), pp. 181–210. DOI: 10.1145/103135.103136.

[43] Glynn Winskel. “Event structures”. In: *Petri Nets: Applications and Relationships to Other Models of Concurrency*. Ed. by W. Brauer, W. Reisig, and G. Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 325–392. DOI: 10.1007/3-540-17906-2_31.

[44] Naling Zhang, Markus Kusano, and Chao Wang. "Dynamic partial order reduction for relaxed memory models". In: *SIGPLAN Not.* 50.6 (June 2015), pp. 250–259. DOI: 10.1145/2813885.2737956.