Model Checking for a Multi-Execution Memory Model

EVGENII MOISEENKO, JetBrains Research, Serbia
MICHALIS KOKOLOGIANNAKIS, MPI-SWS, Germany
VIKTOR VAFEIADIS, MPI-SWS, Germany

Multi-execution memory models, such as Promising and Weakestmo, are an advanced class of weak memory consistency models that justify certain outcomes of a concurrent program by considering multiple candidate executions collectively. While this key characteristic allows them to support effective compilation to hardware models and a wide range of compiler optimizations, it makes reasoning about them substantially more difficult. In particular, we observe that Promising and Weakestmo inhibit effective model checking because they allow some suprisingly weak behaviors that cannot be generated by examining one execution at a time.

We therefore introduce Weakestmo2, a strengthening of Weakestmo by constraining its multi-execution nature, while preserving the important properties of Weakestmo: DRF theorems, compilation to hardware models, and correctness of local program transformations. Our strengthening rules out a class of surprisingly weak program behaviors, which we attempt to characterize with the help of two novel properties: *load buffering race freedom* and *certification locality*. In addition, we develop WMC, a model checker for Weakestmo2 with performance close to that of the best tools for per-execution models.

CCS Concepts: • **Theory of computation → Verification by model checking**; • **Software and its engineering → Semantics**; **Concurrent programming languages**.

Additional Key Words and Phrases: Weak memory models, model checking

ACM Reference Format:
Evgenii Moiseenko, Michalis Kokologiannakis, and Viktor Vafeiadis. 2022. Model Checking for a Multi-Execution Memory Model. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 152 (October 2022), 28 pages. https://doi.org/10.1145/3563315

1 INTRODUCTION

A *weak memory model* is a formal definition of the semantics of shared-memory concurrent programs, which allows more program outcomes (i.e., reachable thread configurations) than can be explained by a straightforward interleaving of the threads of a program. Consider, for instance, the load-buffering (LB) program below. (In our examples, we assume all variables are initialized to 0.)

$$\begin{array}{l|l} r_1 := x \ \textit{// reads 1} & r_2 := y \ \textit{// reads 1} \\ y := 1 & x := r_2 \end{array} \qquad (\text{LB})$$

The annotated outcome, where both threads read the value 1, cannot be explain by simply interleaving the instructions of the two threads, since the first instruction to execute can only read 0 (the initial value). Multicore Arm processors, however, do exhibit this outcome because they often execute independent instructions out of order. For example, Thread 1 may first perform the $y := 1$ write, then Thread 2 can read $y = 1$ and write $x := 1$, and then Thread 1 can finish by reading $x = 1$.

Authors' addresses: Evgenii Moiseenko, JetBrains Research, Serbia, evgeniy.moiseenko@jetbrains.com; Michalis Kokologiannakis, MPI-SWS, Germany, michalis@mpi-sws.org; Viktor Vafeiadis, MPI-SWS, Germany, viktor@mpi-sws.org.

2475-1421/2022/10-ART152
https://doi.org/10.1145/3563315

Most memory models—including all those for hardware architectures—are defined in a *per-execution* style, where each possible program outcome is explained by a single program execution witnessing that outcome. As Batty et al. [2015] have shown, however, this per-execution style is inadequate for defining the semantics of programming languages like C/C++ that strive to provide features with optimal efficiency.

For this reason, advanced language-level memory models (e.g., [Chakraborty et al. 2019; Jagadeesan et al. 2020; Jeffrey et al. 2016; 2022; Kang et al. 2017; Manson et al. 2005; Paviotti et al. 2020; Pichon-Pharabod et al. 2016]) instead adopt a *multi-execution* style, where multiple candidate program executions are used together to justify a given program outcome. For example, Promising [Kang et al. 2017] achieves this by augmenting the regular in-order program execution with an additional step: a promise to execute a future write that is backed up with an additional execution justifying that it is possible to fulfill the promise. Promising explains the LB outcome as follows. Thread 1 first promises to write $y := 1$, which is fulfillable by running the thread to completion. Then Thread 2 reads 1 and writes $x := 1$, and then Thread 1 reads 1, arriving at the desired outcome. The fact that promises have to be justified by other promise-free executions is what makes Promising a multi-execution model. And this aspect of Promising is crucial: removing promise certification would lead to an overly weak model that would allow some clearly undesirable outcomes known as 'out-of-thin-air' outcomes in the literature. Other multi-execution models, such as Weakestmo [Chakraborty et al. 2019], explain the LB outcome in a similar way, but explicitly represent the multiple program executions as an event structure.

While multi-execution models, such as Promising and Weakestmo, can be implemented efficiently on a wide range of hardware architectures [Moiseenko et al. 2020; Podkopaev et al. 2019], they have a significant drawback: they are very difficult to reason about, especially in an automated fashion. Although there are numerous effective automated techniques for verifying programs under per-execution weak memory models (e.g., [Abdulla et al. 2015a; 2018; 2015b; Barnat et al. 2013; Bouajjani et al. 2013; Demsky et al. 2015; Huang et al. 2016; Kokologiannakis et al. 2017; 2019]), there are no automated techniques for reasoning about multi-execution models. Verification of finite-state programs (with loops) is undecidable [Abdulla et al. 2021], and even model checking (i.e., enumerating all possible outcomes) of small loop-free programs is typically intractable.

We believe that the difficulty in automated verification on multi-execution models is largely due to the unconstrained nature of the models' out-of-order execution mechanisms. To make such a model amenable to model checking, one has to constrain the use of multiple executions in two distinct ways: (1) on *when* multiple executions are introduced to explain a certain behavior (i.e., in terms of Promising, when a promise can be made); and (2) on *how much* these multiple executions interact with one another. Naturally, we desire to restrict multi-execution nature as much as possible: additional executions should only be allowed when there is a good reason to do so, and they should not be allowed to diverge too much from one another.

Our first contribution is to propose two properties that constrain multiple executions in the aforementioned ways, namely *load buffering race freedom* and *certification locality* (§2). Besides their use for model checking, these properties rule out certain program outcomes that cannot be observed by any combination of reasonable compiler optimizations on existing hardware platforms, and so may be of independent interest.

Subsequently, we introduce Weakestmo2, a strengthening of Weakestmo [Chakraborty et al. 2019] that satisfies load buffering race freedom and certification locality (§3). Further, we show that Weakestmo2 preserves the soundness of Weakestmo's efficient compilation schemes to hardware-level models shown by Moiseenko et al. [2020] (§3.4) as well as the soundness of local program transformations (§3.5).

Finally, we develop an effective model checking algorithm, WMC, for verifying programs running on Weakestmo2 (§4), and implement it as an extension of the GenMC model checker [Kokologiannakis et al. 2019; 2021]. Our experiments (§5) demonstrate that WMC's performance is superior to that of the few other tools for (per-execution) weak memory models admitting the LB behavior (i.e., the annotated outcome of the LB program), and comparable to the best tools for models that forbid the LB behavior.

2 OVERVIEW

In this section, we recall the basic terminology of weak memory models (§2.1) and motivate our Weakestmo2 model. For the latter, we introduce two properties of multi-execution models that are needed for effective model checking but are not satisfied by Promising and Weakestmo.

Load buffering race freedom (LBRF, §2.2) restricts the multi-execution nature of a model to affect only programs with *load buffering races* (LB races). LBRF is defined analogously to the well-known *data race freedom* (DRF) guarantee [Adve et al. 1996; Manson et al. 2005] replacing the notion of a data race with a stronger novel notion of an LB race.

Certification locality (CL, §2.3) concerns multi-execution models with a certification mechanism and restricts *how much* certification executions can differ from the main execution. CL allows one to determine locally whether a certain load-store reordering is allowed, which disallows certain 'bait-and-switch' behaviors [Jagadeesan et al. 2020].

We conclude this section by explaining how these properties enable effective model checking (§2.4).

2.1 Execution Graphs and Data Race Freedom

In the literature of axiomatic (a.k.a. declarative) per-execution memory models, the possible executions of a program P under a model M are represented as a set of *execution graphs* that correspond to the instructions of P and satisfy M's consistency predicate. Execution graphs consist of:

- a set of nodes, called *events*, which represent the individual operations performed by the program that are relevant for concurrency (e.g., reads R, writes W, fences F), and
- various kinds of directed edges between events, such as:
  - the *program order* (po), relating events in the same thread in their control-flow order as well as initialization events before other events, and depicted as a solid black edge;
  - the *reads-from* (rf) relation, connecting each read to the write it is reading from, and depicted as a dashed green edge from the write to the read;
  - the *happens-before* (hb) order, a subset of $\mathsf{porf} \triangleq (\mathsf{po} \cup \mathsf{rf})^+$ that includes po, capturing ordering due to intra-thread control-flow and inter-thread synchronization. (For simplicity, the examples of this paper do not contain inter-thread synchronization, and so $\mathsf{hb} = \mathsf{po}$.)

The strongest useful model in this framework is *sequential consistency* (SC) [Lamport 1979], which requires that there be a total order $<_{SC}$ among all events of an execution graph extending porf such that each read reads from the most recent same-location write that precedes it in $<_{SC}$. Other models place weaker constraints, with models such as x86-TSO [Owens et al. 2009] and RC11 [Lahav et al. 2017] requiring (among other things) that porf be acyclic.

Figure 1 shows four execution graphs corresponding to the LB program from §1. These graphs are generated by picking every possible value for each read event that matches the value written by the write event to the same location whence the read is reading from. The first three graphs are SC-consistent (and therefore also RC11-consistent). By contrast, the fourth graph, witnessing the "load buffering" behavior, is not RC11-consistent because it contains a porf cycle.

A standard property that is expected of memory models is the DRF_{SC} guarantee [Adve et al. 1996; Manson et al. 2005], which constrains non-SC behaviors to occur only on programs with data

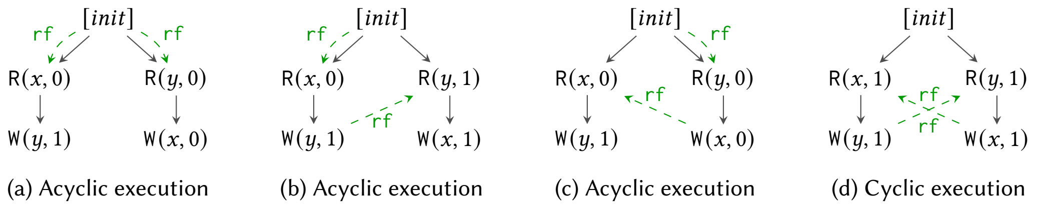


Fig. 1. Execution graphs of LB.

races. We say that two events (of different threads) are *concurrent* if they are not ordered by hb. A pair of events form a *data race* in an execution graph if they are concurrent memory accesses to the same location, at least one of which is a write event. A program P is *data-race-free* under a memory model M if no consistent execution graph of P under M contains a data race.

Definition 2.1 (DRF). A memory model M provides the *data race freedom guarantee* with respect to a stronger memory model M' (written $\mathsf{DRF}_{M'}$) if, for any data-race-free program P under M', its consistent executions under M are exactly the same as under M'.

A memory model providing the $\mathsf{DRF}_{\mathsf{SC}}$ guarantee allows programmers to adopt a defensive programming strategy of avoiding data races (e.g., by using locks), which incurs some performance degradation but relieves them from the need to learn and understand the memory model definition.

2.2 Load Buffering Race Freedom

Load buffering race freedom is analogous to DRF. We say that a po edge between a read and a write is *reorderable* (rpo) if the two accesses are *relaxed* following C11 terminology (i.e., weaker than release/acquire) and there is no fence between them. A reorderable edge signifies that the two events may be executed out of order (e.g., under Promising [Kang et al. 2017], the write may be promised), and thus contribute to a load buffering behavior.

Definition 2.2 (Load buffering race). A pair of events r and w form a *load buffering race (LB race)* in an execution graph if r is a read, w is a concurrent write to the same location, and there is a $\mathtt{rpo} ; (\mathtt{rf} \setminus \mathtt{po}) ; \mathtt{porf}$ path from r to w (i.e., a porf-path starting with a reorderable edge).

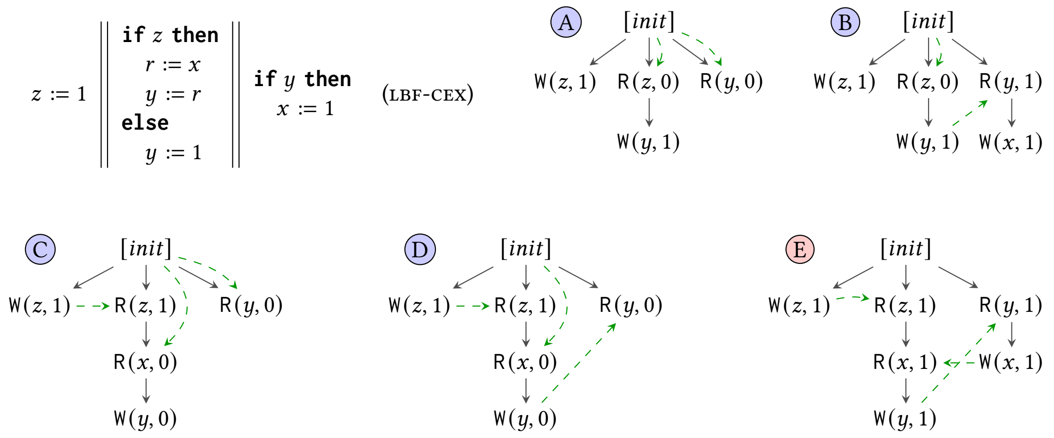
For instance, execution graph (b) of Fig. 1 has a load buffering race between the $\mathtt{R}(x, 0)$ and $\mathtt{W}(x, 1)$ events. Similarly, graph (c) has an LB race between the $\mathtt{R}(y, 0)$ and $\mathtt{W}(y, 1)$ events.

Existence of a porf-path between w and r indicates that the write might depend on the read. In models like RC11, it means that r cannot read from w because that would create a porf-cycle, such as the one in Fig. 1(d). We insist that the first edge along this path is reorderable to rule out cases where an explicit fence has been added to prevent the load-buffering behavior.

Definition 2.3 (LB-race-free program). A program P is *LB-race-free* under a memory model M if no consistent execution graph of P under M contains a load buffering race.

Definition 2.4 (LBRF). A memory model M provides the *load buffering race freedom* guarantee with respect to a stronger memory model M' (written $\mathsf{LBRF}_{M'}$) if, for any LB-race-free program P under M', its consistent executions under M are exactly the same as under M'.

Normally, we take M' to be some standard model that forbids porf-cycles, such as RC11 . Similar to DRF, $\mathsf{LBRF}_{\mathsf{RC11}}$ allows one to program defensively against a model M without even knowing its definition by avoiding LB races. Since absence of LB races is to be checked with respect to RC11, one can use any of the existing tools and methodologies that reason about program correctness



```
z := 1 || if z then     || if y then        (LBF-CEX)
       ||    r := x     ||    x := 1
       ||    y := r     ||
       || else          ||
       ||    y := 1     ||
```

Fig. 2. A program with its RC11-consistent execution graphs and a problematic cyclic execution.

under RC11 (e.g., iGPS, Herd, rcmc, GenMC, etc). Moreover, LB races can easily be removed by making the porf-paths from the racy read to the write start with non-reorderable edges. This can be achieved, for example, by strengthening the access mode of the read to be an acquire or by adding an acquire or release fence right after it. In § 5.4, we report on a script that does so automatically. Based on the results of Ou et al. [2018] and our experience, the run-time overhead incurred by these extra fences is extremely low, if at all perceptible.

2.2.1 LBRF *and Existing Models.* Among the axiomatic per-execution models that allow LB behaviors, the original C11 model [Batty et al. 2011] does not satisfy $\mathsf{LBRF}_{\mathsf{RC11}}$ because it allows out-of-thin-air outcomes and does not even satisfy $\mathsf{DRF}_{\mathsf{SC}}$. Lower-level models that track syntactic dependencies between instructions, such as IMM [Podkopaev et al. 2019], Power [Alglave et al. 2014], and ARM-8 [Pulte et al. 2018], satisfy LBRF wrt. their strengthenings with porf-acyclicity. The additional behaviors they allow over their strengthenings are porf-cycles with at least one po edge from a load to a store being reorderable. By changing the rf edge of that load to read from a prior store, and relying on "receptiveness" of the mapping from programs to executions, we can construct an LB race. As an example of this reasoning, we prove the following theorem:

Theorem 2.5. IMM *satisfies* $\mathsf{LBRF}_{\mathsf{RC11}}$. *(See [Moiseenko et al. 2022, A] for the proof.)*

Among the multi-execution models, Promising [Kang et al. 2017; Lee et al. 2020] and Weakestmo [Chakraborty et al. 2019] do not satisfy $\mathsf{LBRF}_{\mathsf{RC11}}$. To see this, consider the lbf-cex program along with its executions shown in Fig. 2. Although none of its RC11-consistent executions (A, B, C, D) contains an LB race, both Promising and Weakestmo allow the additional execution E where r gets the value 1. This execution can arise in the following manner. First, thread 2 promises the $\mathtt{W}(y, 1)$ store; the promise is allowed because thread 2 can read $z = 0$ and fulfill it. Then, thread 3 executes: it reads $y = 1$ and writes 1 to x. Finally, threads 1 and 2 execute: thread 2 reads $z = 1$ and $x = 1$, and subsequently writes 1 to y thereby fulfilling its promise.

That said, it is fairly straightforward to strengthen Promising and Weakestmo to satisfy LBRF by restricting when promises can be made. The existing condition of Promising requires there to be a thread-local execution that certifies the promise. In addition to that, we can require there to be an execution witnessing an LB race. As we shall shortly see, however, satisfying LBRF alone is not sufficient for effective model checking. We need another locality property that completely forbids non-local certifications that depend on external writes like the $z = 1$ write in the example above.

2.3 Certification Locality

LBRF restricts *when* writes may be promised (i.e., only upon LB races), but not *how*. In models like Promising and Weakestmo (even if artificially strengthened to satisfy LBRF), certifications of outstanding promises can differ a lot from the actual executions whose promises they are certifying. As we will shortly see, this leads to some rather weak outcomes.

We start with an example that does not directly constitute an odd behavior but that is indicative of what can go wrong with non-local certifications.

```
x := 1 || r1 := x  // reads 1 || r2 := y  // reads 1
       || y := r1              || x := r2
```

$$\begin{tikzcd}
 & \left[init\right] \arrow[dl] \arrow[d] \arrow[dr] & \\
\mathtt{W}(x,1) & \mathtt{R}(x,1) \arrow[d] & \mathtt{R}(y,1) \arrow[d] \\
 & \mathtt{W}(y,1) \arrow[ur, dashed] & \mathtt{W}(x,1) \arrow[ul, dashed]
\end{tikzcd}$$

(LB+DEP+W)

The annotated outcome is perfectly valid if Thread 2 gets the value 1 from Thread 1 and propagates it to Thread 3—this outcome is even allowed under SC. What is odd, however, is to justify the outcome by the depicted execution graph, where Threads 2 and 3 read from each other without any flow of information from Thread 1 to either of these threads. The problem is that both Promising and Weakestmo essentially allow this justification. After executing Thread 1, Thread 2 can promise to write 1 to y (by reading from Thread 1). Thread 3 then executes to completion, reading from Thread 2, and writing 1 to x. Finally, Thread 2 continues, reads from Thread 3 and fulfills its promise, thus resulting in the graph above. What went wrong in this execution is that, while the write of Thread 1 was needed to enable the early execution of the write in Thread 2, this dependency is then forgotten in the final execution.

Certification locality forbids exactly this pattern. We say that a write is *non-local* at a certain point in a thread if it neither happens-before that point nor is read by an event happening-before that point. CL requires that every non-local write that is read in the process of certifying a promise also be read by the same thread while issuing the promise and vice versa. In other words, the two executions of the thread issuing a promise should read exactly the same non-local writes between the point the promise is issued (i.e., when the executions start diverging because they read from different writes) and the points where the promises are fulfilled.

To further justify CL, we next show an example of a really weak behavior allowed by Promising and Weakestmo. Consider the LB+LOOP program below and the associated execution graph that reads an arbitrary natural number n into r_1 and r_2.

```
r1 := x              || r2 := y
for i := 0 to r1     || for j := 1 to r2
  y := i + 1         ||   x := j
```

$$\begin{tikzcd}
 & \mathtt{R}(x,n) \arrow[r] & \mathtt{W}(y,1) \arrow[r] & \mathtt{W}(y,2) \arrow[r] & \cdots \arrow[r] & \mathtt{W}(y,n) \arrow[r] \arrow[ddllll, dashed] & \mathtt{W}(y,n+1) \\
\left[init\right] \arrow[ur] \arrow[dr] & & & & & & \\
 & \mathtt{R}(y,n) \arrow[r] & \mathtt{W}(x,1) \arrow[r] & \mathtt{W}(x,2) \arrow[r] & \cdots \arrow[r] & \mathtt{W}(x,n) \arrow[uullll, dashed] &
\end{tikzcd}$$

(LB+LOOP)

The displayed outcome is allowed by Promising (and similarly by Weakestmo) because Thread 1 can initially promise $y := 1$ (which it can trivially fulfill), then Thread 2 can promise $x := 1$ (which it can fulfill by reading $y = 1$ from Thread 1), then Thread 1 can promise $y := 2$ (by reading $x = 1$), then Thread 2 can promise $x := 2$ and so on. Again, the problem in this unbounded execution is the lack of certification locality. Once a thread depends on an external write to justify a promise, it should not "change its mind" and ignore that write in favor of a different one.

2.3.1 Enforcing CL*.* Repairing Promising to satisfy CL is not as straightforward as it is for LBRF because it does not track the exact set of writes read by a thread. We therefore instrument the Promising state by attaching to each promise a set of external writes that must be read before fulfilling the promise. We then check that as long as a thread contains outstanding promises, it can only read from external writes that are recorded in its promise set and, moreover, that when a promise is fulfilled all its attached external writes have been read. We present a suitable definition in [Moiseenko et al. 2022, B]. While our adapted definition achieves certification locality, we have not investigated whether it also satisfies the remaining properties of the original Promising model, namely correctness of compilation and source-to-source transformations.

By contrast, repairing Weakestmo is much easier because the certification runs are available as part of the event structure. To enforce CL, one can simply augment the Weakestmo definition with an axiom that rules out 'bait-and-switch' behaviors. We defer the formal definition of our resulting model, Weakestmo2, to §3. There we also establish three important results about Weakestmo2: it provides the $\mathsf{LBRF}_{\mathsf{RC11}}$ and $\mathsf{DRF}_{\mathsf{SC}}$ guarantees (§3.3), its expected compilation schemes to hardware models are sound (§3.4), and it supports the expected local reorderings and eliminations (§3.5).

2.4 WMC: Effective Model Checking for Multi-Execution Memory Models

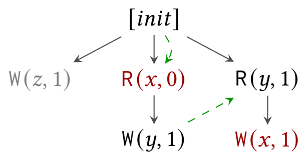
Apart from serving as criteria to rule out certain weak behaviors of multi-execution memory models, LBRF and CL are instrumental in making model checking of programs on such models feasible. We briefly describe how WMC, our model checking algorithm for Weakestmo2, exploits these properties. (We defer the full presentation of WMC to §4.)

Stateless model checkers, such as GenMC [Kokologiannakis et al. 2019], generate all the consistent executions of a given program in an incremental fashion. Starting with the empty execution graph, they add one event at a time in all possible consistent ways. With multi-execution memory models, there are two ways in which events can be added: either by adding events in order or by promising a write out of order. Considering all possible promises, however, is infeasible because of the huge state-space that would need to be explored.

This is where LBRF helps: Promises only need to be considered when an LB race is encountered. WMC therefore first generates the RC11-consistent executions of a program using techniques from the literature. Whenever it discovers an LB race, it uses an additional promising/certification mechanism to explore executions with porf cycles. As an example of how WMC works at a high level, consider the LB program from §1. Under RC11, LB has 3 consistent executions (Figures 1a to 1c), and WMC starts by enumerating those. While doing so, however, it notices that the execution of Fig. 1b contains a load buffering race between events $\mathtt{R}(x, 0)$ and $\mathtt{W}(x, 1)$. Thus, WMC creates the execution of Fig. 1d by *promising* the write $\mathtt{W}(y, 1)$ in Thread 1 (so that $\mathtt{R}(y, 1)$ can read from it), and then by making $\mathtt{R}(x, 0)$ read from $\mathtt{W}(x, 1)$. Subsequently, it *certifies* the promised write in Thread 1, and generates the cyclic execution.

This is exactly the point where CL is useful: It ensures that a promise can quickly be certified or withdrawn by executing only the thread containing the racy read. We illustrate this point with the program below, which is a slight variant of LBF-CEX with the reads of x and z swapped in Thread 2. This program has an acyclic execution with an LB race on the x accesses (shown to the side), and so the model checker will have to consider promising $\mathtt{W}(y, 1)$ in Thread 2.

```
        ||  if x then   ||
        ||    r := z    ||  if y then
z := 1  ||    y := r    ||    x := 1
        ||  else        ||
        ||    y := 1    ||
```



(LB+RDEP)

The model checker will then try to certify the promise by re-executing Thread 2 when it reads 1 for x. The only way to certify the promise is by reading from $\mathtt{W}(z, 1)$. However, depending on the scheduling strategy, this write might not be available at the point when the LB race was detected. Therefore, to avoid missing any executions, the model checker would have to postpone the certification of $\mathtt{W}(y, 1)$ and first explore other parts of the program, an exploration which may itself involve more promises and certifications. This approach is not only complicated to implement correctly, but also rather inefficient because in the fairly common case that the promise cannot be certified, one would wastefully explore all the possible executions of other parts of the program, leading to many blocked explorations.

CL enables a crucial optimization: Since (under CL) certification runs can read only from writes in the porf-prefix of the promise (see §3), the promise can be certified *locally* and *immediately* by re-executing the thread in question. In our example, regardless of whether $\mathtt{W}(z, 1)$ has been added to the graph or not, WMC will only consider the read $\mathtt{R}(z, 0)$, and therefore conclude that the promise $\mathtt{W}(y, 1)$ cannot be certified.

As we show in §5, our technique for promising writes *only* upon detecting an LB race and certifying promised writes *immediately* and *locally* scales much better than the existing techniques that tackle similar memory models. Indeed, the few existing model checkers that handle such models (e.g., [Norris et al. 2013; Pulte et al. 2019]) explore all possible certifiable promises, irrespective of whether they result in additional behaviors. Such wasteful blind exploration of promises is the key factor contributing to their poor performance.

Further, the optimization due to LBRF can also be used to improve by a moderate amount model checking on per-execution models that allow LB behaviors, such as IMM. To demonstrate this, we took HMC [Kokologiannakis et al. 2020], a model checker that operates under IMM, and implemented $\mathrm{HMC}_{\mathsf{LBRF}}$, a version of HMC that leverages LBRF. Similarly to WMC, $\mathrm{HMC}_{\mathsf{LBRF}}$ starts by enumerating RC11-consistent executions, and falls back to dependency tracking only upon detecting an LB race. As we show in §5, $\mathrm{HMC}_{\mathsf{LBRF}}$ outperforms HMC in LB-race-free programs, thereby further showcasing LBRF's usefulness in verification.

3 REPAIRING WEAKESTMO

In this section, we describe how Weakestmo2, our strengthening of Weakestmo, supports LBRF and CL. In what follows, we assume a simplified version of the model, containing only relaxed accesses and fences. For the full model, we refer the reader to Chakraborty et al. [2019] and [Moiseenko et al. 2022, p. C].

Weakestmo is an axiomatic multi-execution memory consistency model. Unlike conventional axiomatic models, which determine the validity of particular outcome based on a consistency predicate on a single execution graph, Weakestmo considers the execution graph consistent if it can be extracted from some consistent *event structure*. Event structures encompass multiple runs of a program in a single graph. That is, event structures can contain several execution branches of the same thread, which are used to model the speculative out-of-order execution of instructions.

Definition 3.1. An event is a tuple $\langle \mathtt{id}, \mathtt{tid}, \mathtt{lab} \rangle$ where $\mathtt{id} \in \mathbb{N}$ is a unique identifier for the event, $\mathtt{tid} \in \mathbb{N}_\perp$ identifies the thread to which the event belongs ($\perp$ for initialization events), and $\mathtt{lab}$ is a *label* of the form: (1) $\mathtt{R}(x, v)$ for a read of $v \in \mathsf{Val}$ from $x \in \mathsf{Loc}$; (2) $\mathtt{W}(x, v)$ for a write of $v \in \mathsf{Val}$ to $x \in \mathsf{Loc}$; (3) $\mathtt{F}$ for a fence.

We write Event for the set of all events. The set of all reads is $\mathsf{R} \triangleq \{\langle i, t, l \rangle \mid l = \mathtt{R}(.)\}$. The sets of all writes and fences are defined analogously. Given an event e, we write $\mathtt{id}(e)$, $\mathtt{tid}(e)$, and $\mathtt{lab}(e)$ to project its components, and $\mathtt{loc}(e)$ and $\mathtt{val}(e)$ to project its location and value respectively.

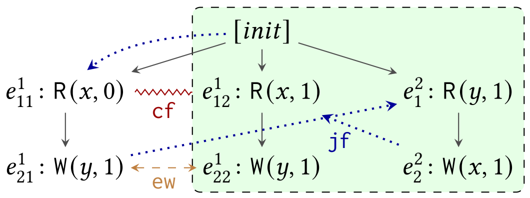


Fig. 3. LB event structure and an extracted execution.

Further, given a relation r, we use $dom(r)$ and $rng(r)$ to denote its domain and codomain, while $r^?$, r^+, and r^* denote its reflexive, transitive, and reflexive-transitive closures, respectively. We write r^{-1} for the inverse of r, $[A]$ for the identity relation $\{(a,a) \mid a \in A\}$, $r_1; r_2$ for the composition of r_1 and r_2: $\{(a,b) \mid \exists c\ (a,c). \in r_1 \wedge (c,b) \in r_2\}$, and $r|_s$ for the restriction of r onto set s: $r|_s \triangleq r \cap s \times s$. When f is a function, we define $=_\mathtt{f} \triangleq \{(a,b) \mid \mathtt{f}(a) = \mathtt{f}(b) \neq \bot\}$.

Definition 3.2. An *event structure* S is a tuple with the following components:

- $\mathsf{E} \subseteq$ Event is a set of events with unique identifiers that includes initialization events, writing 0 to each memory location used by the program.
- $\mathsf{po} \subseteq \mathsf{E} \times \mathsf{E}$ is the *program order*, which captures the order of events of the same thread according to the program's control flow, and orders initialization events before all other events. Because an event structure can contain multiple runs of one thread, po does not have to be total among the events of the same thread. Events of the same thread unordered by po are said to be in *conflict*: $\mathtt{cf} \triangleq =_\mathtt{tid} \setminus (\mathsf{po} \cup \mathsf{po}^{-1})^?$. Two events are in *immediate conflict* if they are in conflict but their predecessors are not: $\mathtt{cf}|_\mathrm{imm} \triangleq \mathtt{cf} \setminus (\mathtt{cf} ; \mathsf{po} \cup \mathsf{po}^{-1} ; \mathtt{cf})$.
- $\mathtt{jf} \subseteq [\mathtt{W}] ; (=_\mathtt{loc} \cap =_\mathtt{val}) ; [\mathtt{R}]$ is the *justified from* relation, which maps each read event to the write event that justifies it. We assume that jf is functional and complete: for each read event there exists a unique write event that justifies it.
- $\mathtt{ew} \subseteq [\mathtt{W}] ; (\mathtt{cf} \cap =_\mathtt{loc} \cap =_\mathtt{val}) ; [\mathtt{W}]$ is the *equal-writes* relation which relates conflicting writes that are considered equal. Its reflexive-transitive closure $\mathtt{ew}^*$ is an equivalence relation on write events.
- $\mathtt{co} \subseteq [\mathtt{W}] ; (=_\mathtt{loc} \setminus \mathtt{ew}^*) ; [\mathtt{W}]$ is the *coherence* order, a strict partial order that relates non-equal write events at the same memory location. Intuitively, it denotes the global order in which operations to the same memory location become visible to all threads.

Given an event structure S, we write $S.X$ to refer to the X component of S. When S is clear from the context, we occasionally omit the "$S.$".

As an example, Fig. 3 depicts an event structure of LB. We can see, for example, that the events $e^1_{11} \to e^1_{21}$ form a branch of the first thread where the load corresponding to the instruction $r_1 := x$ is justified by initial write. A conflicting branch $e^1_{12} \to e^1_{22}$ is shown on the right. Note that we only depict immediate conflict edges. For example, we draw $e^1_{11} \leftrightsquigarrow e^1_{12}$, but not $e^1_{11} \leftrightsquigarrow e^1_{22}$, as the latter can be derived. All reads have a corresponding write that justifies them, i.e., we have, $\mathrm{init} \xrightarrow{\mathtt{jf}} e^1_{11}$ $e^1_{21} \xrightarrow{\mathtt{jf}} e^2_1$, and $e^2_2 \xrightarrow{\mathtt{jf}} e^1_{12}$. The two writes to location y are considered equal $e^1_{21} \xleftrightarrow{\mathtt{ew}} e^1_{22}$.

Definition 3.3. An *execution graph* G is a conflict-free event structure: $G.\mathtt{cf} = \emptyset$.

PROPOSITION 3.4. *For an execution graph G, we have:*

- *$G.\mathsf{po}$ is total on the events of a given thread;*
- *$G.\mathtt{ew}$ is the empty relation;*
- *$G.\mathtt{co}$ is total on same-location writes.*

Next, we discuss how to extract an execution graph from an event structure. To do so, it is not sufficient to just take a conflict-free subset of events. For example, taking the single event $\{e^1_{12}\}$ from the structure in Fig. 3 does not form an execution graph corresponding to the program LB. In addition, we must take all po-predecessors of the event and the writes that justify them. There is, however, one subtle point. Event e^2_1 is justified by e^1_{21}, which is in conflict with e^1_{12}; thus, instead of e^1_{21}, we pick the equivalent write e^1_{22}. To achieve that, we define the derived *reads-from* relation $\mathtt{rf} \triangleq (\mathsf{ew}^* ; \mathtt{jf}) \setminus \mathtt{cf}$ by extending the $\mathtt{jf}$ relation to ew^* equivalence classes.

Definition 3.5. A *justified configuration* C of the event structure S is a subset of its events $C \subseteq S.\mathsf{E}$, s.t.

- C is conflict free: $\mathtt{cf} \cap C \times C = \emptyset$;
- C is closed w.r.t. po-prefixes: $dom(\mathsf{po} ; [C]) \subseteq C$;
- C is $\mathtt{rf}$-complete: $C \cap \mathsf{R} \subseteq rng([C] ; \mathtt{rf})$.

Definition 3.6. An execution graph G is *extracted from* an event structure S, denoted as $S \rhd G$, if $G.\mathsf{E}$ is a justified configuration of S s.t. $G.\mathsf{x} = S.\mathsf{x}|_{G.\mathsf{E}}$ for $\mathsf{x} \in \{\mathsf{po}, \mathsf{ew}, \mathsf{co}\}$ and $G.\mathtt{jf} = G.\mathtt{rf} = S.\mathtt{rf}|_{G.\mathsf{E}}$.

Definition 3.7. The *behavior* of an execution graph G, denoted as $\mathcal{B}(G)$, is a mapping assigning to each location x its final value, that is, the value written by the co-maximal write to x in G.

For example, in Fig. 3 the set of events marked by ⬚ forms a justified configuration and thus induces an execution graph with the behavior $\{x \mapsto 1, y \mapsto 1\}$.

3.1 Weakestmo **Consistency**

To filter out nonsensical event structures, Weakestmo defines a number of *consistency constraints*, which depend on the following auxiliary definitions.

- $\mathtt{hb} \triangleq (\mathsf{po} \cup \mathtt{sw})^+$ — *Happens-before* is the transitive closure of the program order and *synchronizes-with* relations. The latter connects synchronized events. For example, it connects two fences if there exists a $\mathsf{po} ; \mathtt{rf} ; \mathsf{po}$ path between them.
- $\mathtt{ecf} \triangleq (\mathtt{hb}^{-1})^? ; \mathtt{cf} ; \mathtt{hb}^?$ — *Extended conflict* propagates the conflict relation along $\mathtt{hb}$.
- $\mathsf{eco} \triangleq (\mathsf{co} \cup \mathtt{rf} \cup \mathtt{rf}^{-1} ; \mathsf{co})^+$ — *Extended coherence* is almost a total order on accesses to a given location; it orders every pair of such accesses except for equal writes, and reads reading from the same write.
- $\mathtt{jo} \triangleq (\mathtt{jf} \setminus \mathsf{po}) ; (\mathsf{po} \cup \mathtt{jf})^*$ — *Justification order.* The $\mathtt{jo}$ predecessors of an event e are all the writes that cause the event e indirectly through some inter-thread communication. We call these writes the *justification set* of the event e.
- A write event w is a *promise* w.r.t. an event e if $\langle w, e\rangle \in \mathtt{cf} \cap \mathtt{jo}$. Additionally:
  - if $\langle w, e\rangle \in \mathsf{ew}^+ ; \mathsf{po}^?$ then w is a *certified promise*;
  - if $\langle w, e\rangle \notin \mathsf{ew}^+ ; \mathsf{po}^?$ then w is a *pending promise*.

Consider Fig. 3 again. Write event e^1_{21} is a pending promise w.r.t. e^1_{12} and a certified promise w.r.t. e^1_{22}. The later event is exactly the equal write that certifies the promise.

Definition 3.8. Event structure S is Weakestmo-*consistent* if the following conditions hold.

- $\mathtt{ecf}$ is irreflexive. (NON-CONTRADICTORY)
- $\mathtt{jf} \cap \mathtt{ecf} = \emptyset$ (WELL-JUSTIFIED)
- $\mathsf{po} \cup \mathtt{jf}$ is acyclic (NO-THIN-AIR)
- $\mathtt{hb} ; \mathsf{eco}^?$ is irreflexive. (COHERENT)
- $[\mathsf{F}] ; \mathsf{po} ; \mathsf{ew}^+ \subseteq \mathsf{po}$ (WELL-FENCED)
- $\mathtt{cf} \cap \mathtt{jo} \subseteq \mathsf{ew}^+ ; (\mathsf{po} \cup \mathsf{po}^{-1})^?$ (CERTIFIED)

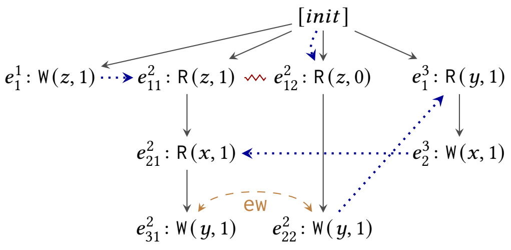


Fig. 4. Weakestmo2 **inconsistent** event structure of LBF-CEX.

- $\mathtt{ew} \subseteq (\mathtt{cf} \cap (\mathtt{jo} \cup \mathtt{jo}^{-1}))^+$ (GROUNDED)

The first two constraints forbid nonsensical event structures where some event either is in conflict with itself, or justifies a conflicting event. NO-THIN-AIR prevents the values of reads to appear out-of-thin-air. COHERENT enforces the *coherence* property at the level of the whole event structure[1]. The last three axioms are related to promises and certification: WELL-FENCED prevents promises to be issued across fences; CERTIFIED ensures that all promises are eventually certified; and GROUNDED guarantees that the ew relation is only used to certify promises, and not to link arbitrary writes.

Definition 3.9. Execution graph G is Weakestmo-*consistent* if there exists Weakestmo-consistent event structure S s.t. G can be extracted from S.[2]

3.2 Weakestmo2 **Consistency**

As already mentioned in §2.2.1, Weakestmo-consistency guarantees neither $\mathsf{LBRF}_{\mathsf{RC11}}$ nor CL. For example, Figure 4 depicts a Weakestmo-consistent event structure of LBF-CEX that justifies the weak outcome $r_1 = 1$, which is forbidden by $\mathsf{LBRF}_{\mathsf{RC11}}$.

In order to get the cyclic execution of LBF-CEX, Thread 2 first issues the promise e_{22}^2. Through Thread 3, this promise justifies another branch of Thread 2, namely $e_{11}^2 \rightarrow e_{21}^2 \rightarrow e_{31}^2$, and can be certified thanks to the equivalent write e_{31}^2. The problem is that in the two branches of Thread 2, the read of z is justified by two different writes. In other words, Thread 2 *baits* other threads with the promise, assuming that read of z gets value 0, but then *switches* by picking value 1 for this read.

To forbid this kind of behavior, we place an additional consistency constraint enforcing CL. We say that a write w *externally justifies* a read r if it justifies r and does not happen-before r nor justifies some other read that happens before r. For example, in Fig. 4, e_1^1 externally justifies e_{11}^2 and e_2^3 externally justifies e_{21}^2. We require that whenever an event structure contains two conflicting branches due to some promise, the external justifications of the two branches agree modulo the external justification that created the conflict between the branches.

Definition 3.10. An event structure S is Weakestmo2-*consistent* if it is Weakestmo-consistent and also:

- $(\mathtt{jf} \setminus (\mathtt{jf}^? ; \mathtt{hb})) ; \mathtt{po} ; \mathtt{ew} \subseteq \mathtt{jf} ; (\mathtt{po} \cup \mathtt{lbpat})$ (NO-BAIT-AND-SWITCH)

where $\mathtt{lbpat} \triangleq \mathtt{cf}_{\mathtt{imm}} ; [\mathit{rng}(\mathtt{jf} \cap (\mathtt{jf}^? ; \mathtt{hb}))] ; \mathtt{po}$ denotes the *load buffering pattern.*

[1]To see why coherence is enforced for the whole event structure rather than on per-execution basis consult Chakraborty et al. [2019, §2.3].
[2]The full version of the model (see ??) filters out executions that violate atomicity of read-modify-write operations or sequential consistency of sc accesses.

$$\begin{tikzcd}
r : \mathtt{R}(x,\alpha) \arrow[r, squiggly, dash] \arrow[dd] & r' : \mathtt{R}(x,\beta) \arrow[d] & \\
 & e_{i+1} : \mathtt{R}(z,\delta) & \\
p : \mathtt{W}(y,\gamma) \arrow[rr, bend right=10, "\mathtt{jo}"] & & w : \mathtt{W}(z,\delta) \arrow[ul, dotted, bend right, "\mathtt{jf}"']
\end{tikzcd}$$

(a) e_{i+1} is in the middle of the certification branch.

$$\begin{tikzcd}
r : \mathtt{R}(x,\alpha) \arrow[r, squiggly, dash] \arrow[d] & e_{i+1} : \mathtt{R}(x,\beta) \\
p : \mathtt{W}(y,\gamma) \arrow[r, bend right=10, "\mathtt{jo}"] & w : \mathtt{W}(x,\beta) \arrow[u, dotted, bend right, "\mathtt{jf}"']
\end{tikzcd}$$

(b) e_{i+1} is the first event of the certification branch.

Fig. 5. Illustration to the proof of the LBRF theorem.

An execution graph is Weakestmo2-*consistent* if it can be extracted from some Weakestmo2-consistent event structure.

The NO-BAIT-AND-SWITCH axiom requires that external justifications of one branch either also justify some read in the conflicting branch ($\mathtt{jf}\,;\mathsf{po}$) or be the very reason why the branch was created ($\mathtt{jf}$; $\mathtt{lbpat}$): the externally justified read should be in immediate conflict with a non-externally justified read heading the other branch.

Due to this axiom, the event structure of Fig. 4 is not Weakestmo2-consistent because event e_2^3 externally justifies e_{21}^2 but does justify any event po-before e_{22}^2. In contrast, the event structure of Fig. 3 is Weakestmo2-consistent. The only relevant external justification is that of e_{12}^1 by e_2^2, which is allowed because e_{12}^1 is in immediate conflict with e_{11}^1, which is po-before e_{21}^1.

3.3 Load Buffering and Data Race Freedom

We recall the definition of RC11-consistency and show that Weakestmo2 satisfies $\mathsf{LBRF}_{\mathsf{RC11}}$.

Definition 3.11. An execution graph G is RC11-*consistent* if the following hold:

- $\mathsf{po} \cup \mathtt{rf}$ is acyclic (NO-THIN-AIR)
- $\mathtt{hb}\,;\mathtt{eco}^?$ is irreflexive. (COHERENT)

THEOREM 3.12. Weakestmo2 *satisfies* $\mathsf{LBRF}_{\mathsf{RC11}}$. *(See [Moiseenko et al. 2022, p. D] for the full proof.)*

PROOF SKETCH. We introduce a subclass of *promise-free* event structures for which $\mathtt{cf} \cap \mathtt{jo} = \emptyset$ holds. It is easy to show that for Weakestmo2-consistent promise-free event structure S justified-from relation coincides with reads-from relation: $S.\mathtt{jf} = S.\mathtt{rf}$. Therefore every extracted execution of a consistent promise-free event structure is RC11-consistent, since $\mathsf{po} \cup \mathtt{rf}$ acyclicity follows immediately from $\mathsf{po} \cup \mathtt{jf}$ acyclicity. Thus it suffices to show that every Weakestmo2-consistent event structure of an LB-race-free program is promise-free.

The latter can be shown by induction on the construction of an event structure. That is, given a Weakestmo2-consistent event structure S one can consider a sequence of events $\{e_1, \ldots, e_n\}$ of S ordered according to some total order extending $(S.\mathsf{po} \cup S.\mathtt{jf})^*$. It is possible to construct S step-by-step by adding single event at each step. Then by induction we can show that each event structure S_i obtained on i-th step is promise-free.

Trivially, empty event structure S_0 is promise free. Also it can be shown that if the event added on $i + 1$ step e_{i+1} is a write or a fence, then the relation $\mathtt{cf} \cap \mathtt{jo}$ cannot increase and thus the event structure remains promise-free. The only non-trivial case is when e_{i+1} is a read event.

This situation is depicted in Fig. 5a. We have a newly added read event e_{i+1} and a promise p. Read events r and r' are the first events at which two branches of the event structure diverge and become conflicting. The constraint NO-BAIT-AND-SWITCH guarantees that the read e_{i+1} cannot observe promise p in the middle of the certification branch, thus it has to be that $e_{i+1} = r'$ (see Fig. 5b).

Then it is easy to see that events r and w form a load-buffering race. Both of these events belong to the event structure S_i obtained on the previous step. Because of our inductive assumption, S_i is promise-free. Thus we can extract an RC11-consistent execution containing a load-buffering race, contradicting our assumption that P is LB-race-free under RC11. Therefore, it has to be that S_{i+1} remains promise-free. □

Composing $\mathsf{LBRF}_{\mathsf{RC11}}$ with RC11's $\mathsf{DRF}_{\mathsf{SC}}$ theorem, we get $\mathsf{DRF}_{\mathsf{SC}}$ for Weakestmo2.

Corollary 3.13. Weakestmo2 *satisfies* $\mathsf{DRF}_{\mathsf{SC}}$.

3.4 Soundness of Compilation Mappings

One of the main objectives of the advanced multi-execution weak memory models is to enable efficient compilation mappings to the hardware architectures. In this section, we show that our modification of the Weakestmo preserves this property. We prove soundness of the optimal compilation schemes, i.e., those that do not require to insert fences or fake dependencies when compiling relaxed accesses, from Weakestmo2 to memory models of x86, ARMv7, ARMv8, and POWER.

To achieve this goal, we adjust the proof of the compilation correctness for Weakestmo by Moiseenko et al. [2020]. The proof uses the IMM model as a mediator between Weakestmo and the hardware models. Since the correctness of compilation mappings from IMM to hardware models is already established by Podkopaev et al. [2019], it suffices to show correctness of compilation from Weakestmo2 to IMM, which boils down to the following statement [Moiseenko et al. 2020, § 2.3].

Theorem 3.14. *Let P be a program, and G be an* IMM*-consistent execution graph of P. Then, there exists an* Weakestmo2*-consistent event structure S of P such that $S \rhd G$.*
(See [Moiseenko et al. 2022, F] for the full proof.)

Proof sketch. To prove the theorem, following the original proof, we construct the required event structure S step by step following a *traversal* of the IMM graph G [Podkopaev et al. 2019, § 6.2]. Traversal of the graph G induces operational small-step semantics $G \vdash TC \xrightarrow{e} TC'$ where TC and TC' are *traversal configurations* and e is an event being traversed. Traversal configuration is a tuple $\langle C, I \rangle$, where $C \subseteq G.\mathsf{E}$ is a set of *covered events* and $I \subseteq G.\mathsf{W}$ is a set of *issued writes*.

Intuitively, covering an event corresponds to in-order execution of an instruction of the program, while issuing a write corresponds to our-of-order speculative execution of some store. An event can be covered whenever (i) all of its po predecessors are covered and (ii) it is already issued or it reads from an issued write. In order to issue a write, one must first issue all the writes of other threads on which it depends via the preserved program order ppo. These constraints can be manifested as the following invariants of the traversal configuration $\langle C, I \rangle$:

$$dom(\mathsf{po}\,;[C]) \subseteq C \qquad C \cap \mathsf{W} \subseteq I \qquad dom(\mathsf{rf}\,;[C]) \subseteq I \qquad dom((\mathsf{rf} \setminus \mathsf{po})\,;\mathsf{ppo}\,;[I]) \subseteq I$$

Giving the operational semantics of traversal $G \vdash TC \xrightarrow{e} TC'$, and the operational semantics of event structure construction $S \xrightarrow{e} S'$ the proof then proceeds using the standard *simulation* argument. As such, the main challenge of our modification of the proof was to show that the new axiom no-bait-and-switch is preserved during the simulation. It turned out that in order to ensure that we need to slightly modify the construction from Moiseenko et al. [2020].

We demonstrate the problem with the original construction and our key idea on how to repair it with an example. Consider program lb-imm in Fig. 6. Its annotated outcome is allowed by IMM and can be seen as a result of reordering the syntactically independent instructions of Thread 1.

To generate this outcome, the first step of the traversal issues $\mathtt{W}(a, 1)$ in Thread 1. To simulate this action, we create the branch $e^1_{11} \to e^1_{21} \to e^1_{31} \to e^1_{41} \to e^1_{51}$ (see Fig. 6) using the *receptiveness* property [Podkopaev et al. 2019, § 6.4]. Receptiveness allows us to pick arbitrary values for intermediate

$$\left.\begin{array}{l} r_1 := z \ /\!/ \ 1 \\ b := x * y \\ a := 1 \end{array} \right\| \left. \begin{array}{l} r_2 := a \ /\!/ \ 1 \\ x := r_2 \\ y := r_2 \end{array} \right\| \begin{array}{l} r_3 := b \ /\!/ \ 1 \\ z := r_3 \end{array}$$

(LB-IMM)

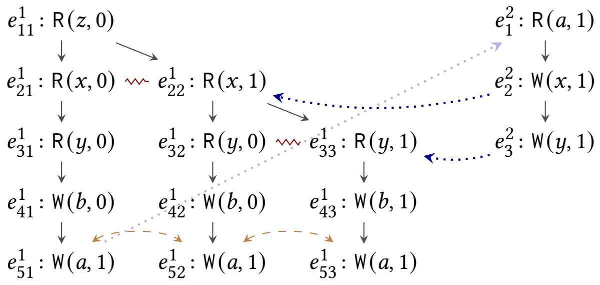


Fig. 6. LB-IMM and its partial Weakestmo2-consistent event structure.

read events in the branch if there is no dependency from these reads to the issued write. For each such read, we choose some "stable" justification write [Moiseenko et al. 2020, § 4.3.1]. In this case, the stable justification writes happen to be the initialization writes (omitted on Fig. 6 for brevity).

The next steps in the traversal cover R$(a, 1)$ and then issue and immediately cover W$(x, 1)$ and W$(y, 1)$ in Thread 2. To match these steps, we add events $e^2_1 \rightarrow e^2_2 \rightarrow e^2_3$ to the event structure.

Subsequently, the traversal issues W$(b, 1)$. At this point, the construction of Moiseenko et al. [2020] adds a branch $e^1_{22} \rightarrow e^1_{33} \rightarrow e^1_{43} \rightarrow e^1_{53}$. It does so by picking suitable justification writes *for all reads* on which the issued write W$(b, 1)$ depends (via ppo). However, changing justification for multiple reads at once in the new branch violates NO-BAIT-AND-SWITCH.

We repair the construction by showing that we can replace justification writes for G.ppo-preceding reads *incrementally* one-by-one, constructing a series of certification branches, as shown in Fig. 6. That is, we first construct the intermediate branch $e^1_{22} \rightarrow e^1_{32} \rightarrow e^1_{42} \rightarrow e^1_{52}$. Doing so satisfies NO-BAIT-AND-SWITCH because the new branch differs from the previous branch only at the point of immediate conflict: $e^1_{21} \leftrightsquigarrow e^1_{22}$. Starting from this intermediate branch, it becomes possible to add the required branch $e^1_{33} \rightarrow e^1_{43} \rightarrow e^1_{53}$ which now has the event e^1_{43} matching the issued write W$(b, 1)$.

The remaining part of the simulation process is not shown in Fig. 6 because it proceeds unchanged compared to Moiseenko et al. [2020]: first R$(b, 1)$ is covered, then W$(z, 1)$ is issued and covered, and finally all the events of Thread 1 are covered. The corresponding events are added to the event structure in a straightforward way to match these traversal steps, arriving at the final execution graph justifying the annotated outcome. □

3.5 Soundness of Program Transformations

Another important objective of the advanced weak memory models is to justify source-to-source program transformations applied by optimizing compilers. We next discuss the implications of strengthening Weakestmo for their soundness. A transformation *tr* from a source program P_{src} to a target program P_{tgt} is *sound* if does not add any new behaviors. That is, for every Weakestmo2-consistent execution G_{tgt} of P_{tgt}, there exists some Weakestmo2-consistent execution G_{src} of P_{src} with the same behavior: $\mathcal{B}(G_{src}) = \mathcal{B}(G_{\mathsf{tgt}})$.

In [Moiseenko et al. 2022, G], we took the results of Chakraborty et al. [2019, §6.2] for the original version of Weakestmo and showed that all sound **reorderings** and **eliminations** of relaxed loads and stores **remain sound** for Weakestmo2. As an example of our reasoning we give a proof sketch for the soundness of the load/store reordering.

THEOREM 3.15. *The reordering of two adjacent independent instructions* $\mathsf{a} = (r_1 := x)$ *and* $\mathsf{b} = (y := r_2)$ *is a sound source-to-source transformation.*

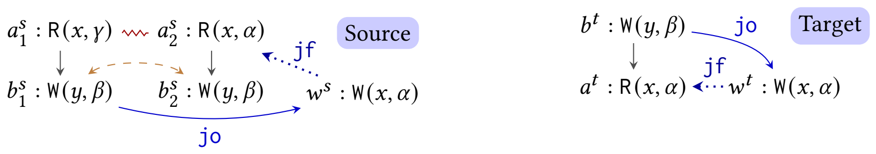

Source
Target
$a^s_1 : \mathtt{R}(x, \gamma)$
$a^s_2 : \mathtt{R}(x, \alpha)$
$b^s_1 : \mathtt{W}(y, \beta)$
$b^s_2 : \mathtt{W}(y, \beta)$
$w^s : \mathtt{W}(x, \alpha)$
jf
jo
$b^t : \mathtt{W}(y, \beta)$
$a^t : \mathtt{R}(x, \alpha)$
$w^t : \mathtt{W}(x, \alpha)$
jo
jf

Fig. 7. A fragment of the event structure construction that justifies load/store reordering.

Proof sketch. The proof proceeds by induction on the construction of the target event structure using the simulation argument of Chakraborty et al. [2019, §F]. Given a Weakestmo2-consistent execution graph $G_{\mathtt{tgt}}$ of $P_{\mathtt{tgt}}$, we consider event structure $S_{\mathtt{tgt}}$, s.t. $S_{\mathtt{tgt}} \rhd G_{\mathtt{tgt}}$, and we built it incrementally from the initial event structure: $S_{\mathrm{init}}(P_{\mathtt{tgt}}) \longrightarrow^* S_{\mathtt{tgt}}$. Following these steps, we will construct $S_{\mathtt{src}}$ and the required graph $G_{\mathtt{src}}$.

To simulate the target event structure construction step $S_{\mathtt{tgt}} \xrightarrow{e} S'_{\mathtt{tgt}}$, if the event added, e, is **not** a result of executing instruction b or a, the source event structure can be augmented by adding the same event $S_{\mathtt{src}} \xrightarrow{e} S'_{\mathtt{src}}$.

Otherwise, let us consult Fig. 7. It depicts a fragment of the source (on the left) and target (on the right) event structures. The target event structure construction adds the event $b^t : \mathtt{W}(y, \beta)$ corresponding to the instruction b. In the source, however, one has to execute instruction a first. In doing so, it might be not possible to add an event with the label $\mathtt{R}(x, \alpha)$ that will be added later in the target event structure. The corresponding justifying write event might have yet been added to the source event structure because it may, in turn, depend on the write event added as a result of executing b itself (see events b^t and w^t in the target event structure).

The construction of Chakraborty et al. [2019, §F] therefore creates an auxiliary execution branch in the source event structure consisting of events a^s_1 and b^s_1. The construction justifies the $a^s_1 : \mathtt{R}(x, \gamma)$ event by choosing the $S_{\mathtt{src}}.\mathtt{co}$-maximal non-conflicting write from $S_{\mathtt{src}}.\mathtt{jf}^? \,;S_{\mathtt{src}}.\mathtt{hb}$ prefix of a^s_1. Since instructions a and b are assumed to be independent, the choice of the value γ for the read a^s_1 cannot affect the value of the write $b^s_1 : \mathtt{W}(y, \beta)$.

The construction then proceeds by adding events to the target and source event structures until the target reaches the event a^t. At this point, the construction adds another branch to the source consisting of events a^s_2 and b^s_2. Now, the event a^s_2 can have the required label $\mathtt{R}(x, \alpha)$ because there is already a justifying event w^s in the source event structure matching the target's justification event w^t. Finally, the write b^s_2 is announced to be equal to b^s_1.

What remains to be shown is that the two conflicting branches $a^s_1 \to b^s_1$ and $a^s_2 \to b^s_2$ satisfy the axiom NO-BAIT-AND-SWITCH. Indeed, they only differ at the point of the immediate conflict, and moreover, the read a^s_1 is justified from a $(S_{\mathtt{src}}.\mathtt{jf}^? \,; S_{\mathtt{src}}.\mathtt{hb})$-preceding write. □

4 WMC: WEAKESTMO2 MODEL CHECKING

In this section, we present WMC, our model checking algorithm for Weakestmo2, which we build on top of GenMC [Kokologiannakis et al. 2019], an existing state-of-the-art, open-source stateless model checker for RC11 programs. Our algorithm is largely parametric in the underlying memory model, and so can in principle be adapted to other multi-execution memory models satisfying LBRF and CL. We proceed with a brief description of how GenMC operates under RC11 (§4.1), and then describe our extensions for Weakestmo2 (§4.2).

Algorithm 1 Main exploration algorithm

```
 1: procedure Visit($P, G, \Pi$)
 2:   if $\neg\mathsf{cons}_{\mathrm{RC11}\setminus\mathsf{porf}}(G)$ then return
 3:   switch $a \leftarrow \mathsf{next}_{P,\Pi}(G)$ do
 4:     case $a = \bot$
 5:       if $\Pi = \emptyset$ then output "Exec OK"
 6:     case $a \in \mathtt{error}$
 7:       exit("Erroneous execution")
 8:     case $a \in \mathsf{R}$
 9:       for $w \in$ GetRFs($G, \Pi, a$) do
10:         Visit($P$, SetRF($G, w, a$), $\Pi$)
11:     case $a \in \mathsf{W}$
12:       $\Pi' \leftarrow \{\langle w, G_w\rangle \in \Pi \mid w \neq a\}$
13:       Visit($P, G, \Pi'$)
14:       VisitRevisits($P, G, \Pi, \Pi', a$)
15:     otherwise Visit($P, G, \Pi$)

 1: function GetRFs($G, \Pi, r$)
 2:   $W \leftarrow G.\mathsf{W}_{\mathtt{loc}(r)}$
 3:   if $\Pi \neq \emptyset$ then
 4:     $L \leftarrow \{w' \mid \exists \langle w, G'\rangle \in \Pi.$
            $\langle w', w\rangle \in G'.\mathtt{rf}^?; G.\mathtt{hb}\}$
 5:     $W \leftarrow W \cap L$
 6:   return $W$
```

Algorithm 2 Exploration of revisits

```
 1: procedure VisitRevisits($P, G, \Pi, \Pi', a$)
 2:   if $\Pi' \neq \emptyset$ then $\langle Rs, \Pi_c\rangle \leftarrow \langle\emptyset, \emptyset\rangle$          ▷ In cert
 3:   else if $\Pi \neq \emptyset$ then                                ▷ Cert OK
 4:     $\langle Rs, \Pi_c\rangle \leftarrow \langle$CertRevs($G, a$)$, \{a\}\rangle$
 5:   else                                                       ▷ Normal exec
 6:     $\langle Rs, \Pi_c\rangle \leftarrow \langle$GetRevs($G, a$)$, \emptyset\rangle$
 7:   for $\langle w, r\rangle \in Rs$ do
 8:     $G' \leftarrow$ Restrict($G, r, w$)$\setminus rng([r]; G.\mathsf{po})$
 9:     $\Pi' \leftarrow$ Promises($G, r, \{w\} \cup \Pi_c \cup$ InCyc($G, r$))
10:     Visit($P, G', \Pi'$)

 1: function GetRevs($G, a$)
 2:   return $\{\langle a, r\rangle \mid r \in (G.\mathsf{T}_{\mathtt{loc}(a)} \setminus dom(G.\mathsf{po}\mathtt{rf}; [a]))$
                  $\cup\, dom(G.\mathtt{lbrace}\,; [a])\}$

 1: function Promises($G, r, S$)
 2:   return $\{\langle w', G\rangle \mid \langle r, w'\rangle \in G.\mathsf{po}\,; [dom(G.\mathtt{rf}\mathsf{e})]$
                  $\wedge\, w' \in dom(G.\mathsf{po}\mathtt{rf}; [S])\}$

 1: function CertRevs($G, e$)
 2:   return $\{\langle r, w\rangle \mid \langle r, e\rangle \in \mathsf{po} \wedge G.\mathtt{rf}; [r] \subseteq G.\mathtt{rf}^?; G.\mathtt{hb}$
                  $\wedge w \in G.\mathsf{W}_{\mathtt{loc}(r)}\} \,\cup$
              $\bigcup_{w \in dom([\mathsf{W}]; G.\mathsf{po}; [e])}$ GetRevs($G, T, w$)
```

4.1 GenMC: Model Checking under RC11

GenMC, like other *dynamic partial order reduction* (DPOR) algorithms [Abdulla et al. 2014; Flanagan et al. 2005], verifies a program by enumerating its executions one at a time, while recording alternative exploration options along the way. This high-level procedure is depicted in Algorithm 1. (The highlighted code represents our extensions for Weakestmo2 and can be ignored for now.)

GenMC's Visit procedure explores all consistent executions of a program P under RC11[3] recursively. During the exploration, Visit maintains the current exploration graph G which is augmented with a *revisit set* $G.\mathsf{T}$ that records all reads in G whose reads-from edge can be changed. Initially, Visit is called with an empty execution graph.

At each step, as long as the current graph remains consistent (Line 2), Visit picks the next event to add, and adds it to the graph using the next function (Line 3).

The role of next is twofold: it schedules a thread t and also adds the next event of t in the graph. If no thread can be scheduled (e.g., if all threads are finished), then the execution is complete, and next returns $\bot$ (Line 4). If an error is encountered (e.g., if an assertion in the program is violated), next returns the error token `error` (Line 7). Otherwise, it returns the event it added a.

If a is a read, Visit has to consider all possible `rf` edges for it. To that end, for each possible `rf` option w (given by GetRFs), Visit recursively calls itself with the graph recording that a read from w (Lines 9 and 10).

If a is a write, apart from simply recursing further (Line 13), Visit has to also check whether w can *revisit* any of the existing reads in G. This is necessary because, when a read r is added to the graph, it may well be the case that some write from which r could also read from has not yet been

[3]Adapting the algorithm for a different model m merely requires changing the consistency check in Line 2 of Algorithm 1.

added to the graph. Thus, whenever Visit adds a write a, it also checks whether any of the existing reads can be revisited to read from a, and explores these options via VisitRevisits (Line 14).

Otherwise, Visit simply recurses further.

The VisitRevisits Procedure. The calculation of revisitable reads is performed by VisitRevisits, by means of GetRevs. As can be seen in Algorithm 2, GetRevs returns a set where a is paired with all revisitable reads to the same location that are not in its porf prefix.

Subsequently (and in accordance to standard DPOR approaches), for each such revisit pair $\langle r, w\rangle$, VisitRevisits first restricts G (Line 8) so that it only contains the events that were added before r, as well as the events that are porf-before w (as w was added after r). Analogously, it restricts $G.\mathsf{T}$ so that it only contains reads added before (and including) r. Finally, VisitRevisits simply calls Visit recursively to generate the corresponding executions.

GenMC: An Example. Let us now illustrate how GenMC works with an example. Consider a variant of the load buffering program from §1 with no dependencies between instructions.

$$\begin{array}{l||l} r_1 := x & r_2 := y \\ y := 1 & x := 1 \end{array} \qquad \text{(LB-NODEP)}$$

We choose this example because it has 3 consistent executions under RC11 (Figures 1a to 1c) and one additional execution under Weakestmo2. Running GenMC on LB-NODEP highlights the difficulties arising when trying to enumerate the executions of programs with porf cycles. In the presentation below, we omit the values read by reads in the graphs, as they can be deduced from the rf edges.

GenMC starts with an empty graph and then adds two events corresponding to $r_1 := x$ and $y := 1$, respectively, as can be seen below.

$$\begin{tikzcd}
\textcircled{0}\ [init] & & \textcircled{1}\ [init] \arrow[d, shift right, dashed, "rf"'] \arrow[d, shift left] & & \textcircled{2}\ [init] \arrow[d, shift right, dashed, "rf"'] \arrow[d, shift left] \\
& \rightsquigarrow & \mathsf{R}(x) & \rightsquigarrow & \mathsf{R}(x) \arrow[d] \\
& & & & \mathsf{W}(y, 1)
\end{tikzcd}$$

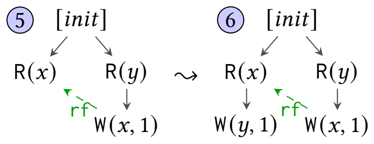
Note that, because there are no other reads-from options for $\mathsf{R}(x)$ except for 0, the loop in Line 9 in Visit will only add one child node to the recursion tree. Similarly, because there are no revisit options for $\mathsf{W}(y, 1)$, VisitRevisits is not executed, and GenMC only calls Visit in Line 13 for the newly added write.

However, when the read event corresponding to $r_2 := y$ is added in the next step, Visit starts two recursively explorations: one where $\mathsf{R}(y)$ reads 0, and an alternative one where it reads from $\mathsf{W}(y, 1)$. Let us assume that Visit first proceeds with the one where $\mathsf{R}(y)$ reads 0.

$$\begin{tikzcd}
\textcircled{2}\ [init] \arrow[d, shift right, dashed, "rf"'] \arrow[d, shift left] & & & \textcircled{3}\ [init] \arrow[dl, shift right, dashed, "rf"'] \arrow[dl, shift left] \arrow[dr, shift left, dashed, "rf"] \arrow[dr, shift right] & & & & \textcircled{4}\ [init] \arrow[dl, shift right, dashed, "rf"'] \arrow[dl, shift left] \arrow[dr, shift left, dashed, "rf"] \arrow[dr, shift right] & \\
\mathsf{R}(x) \arrow[d] & \rightsquigarrow & \mathsf{R}(x) \arrow[d] & & \mathsf{R}(y) & \rightsquigarrow & \mathsf{R}(x) \arrow[d] & & \mathsf{R}(y) \arrow[d] \\
\mathsf{W}(y, 1) & & \mathsf{W}(y, 1) & & & & \mathsf{W}(y, 1) & & \mathsf{W}(x, 1)
\end{tikzcd}$$

In a similar manner, when the write corresponding to $x := 1$ is added to the graph, Visit will initiate two recursive explorations: one where $\mathsf{W}(x, 1)$ does not revisit any reads, and one where it revisits $\mathsf{R}(x)$. Indeed, because $\mathsf{R}(x)$ is *not* porf-before $\mathsf{W}(x, 1)$, it will be considered by GetRevs, and VisitRevisits will initiate a recursive exploration.

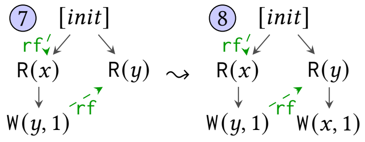
In the non-revisiting case, the graph is complete (corresponding to Fig. 1a), so let us focus on the revisiting case. The graph created for the revisiting exploration is graph $\textcircled{5}$ below.



In effect, this graph models a scenario where the W$(x, 1)$ was added just before the R(x): the only events that are present in the graph are the ones added before R(x) (i.e., the read itself), as well as those that are absolutely necessary in order to trigger W$(x, 1)$ (here, its po-predecessor). Had we only kept the events that were added before R(x) when revisiting (as Visit does in the read case), W$(x, 1)$ would not existed upon restriction, and R(x) would not have been able to read from it.

Continuing with the revisiting case, W$(y, 1)$ is added once again to the graph. This time, however, it does not revisit R(y), as the latter is both porf-before W$(y, 1)$, and also not revisitable (recall that T is restricted to only contain events that were added before R(x)). The graph is now complete again (corresponding to Fig. 1c), and thus Visit backtracks and explores the alternative rf for R(y).

The final exploration can be seen below. Since the recursive call corresponding to ⑦ was initiated by the addition of R(y), graph ⑦ is identical to ③ with the only exception being R(y)'s rf edge.



In the final step of the algorithm, Visit adds W$(x, 1)$ again in the graph. Since R(x) is porf before W$(x, 1)$, no reads are considered by GetRevs, and the final exploration is concluded (cf. Fig. 1b).

4.2 WMC: Model Checking under Weakestmo2

We just saw how GenMC explores all three RC11-consistent executions of the LB program. When it comes to generating the execution of Fig. 1d, however, GenMC fails, for two major reasons.

First, GetRevs does not return reads that are porf-before the revisiting write. As such, even though W$(y, 1)$ (resp. W$(x, 1)$) could revisit R(y) (resp. R(x)) in executions ⑥ and ⑧ to obtain the cyclic execution, that was impossible due to a porf-path between the read and the write.

Second, revisiting porf-earlier reads (which would seemingly solve the first issue above) is not enough on its own, as such reads may be non-revisitable (as e.g., R(y) in ⑥).

We next show how WMC overcomes these difficulties and generates the execution of Fig. 1d, using the ideas of §2.4. Our WMC extensions are highlighted in Algorithms 1 and 2.

4.2.1 WMC: Overview. In order to generate LB behaviors, the first thing that needs to be changed is the function GetRevs(G, a). Besides the revisitable reads that are not porf-before a (as in GenMC's case), WMC also returns any reads that are porf-before a, as long as they are in an LB race with a:

$$\mathtt{lbrace} \triangleq ([\mathsf{R}] ; =_{\mathtt{loc}} ; [\mathsf{W}]) \cap (G.\mathtt{rpo} ; (G.\mathtt{rf} \setminus G.\mathtt{po}) ; G.\mathtt{porf})) \setminus (\mathtt{hb} \cup \mathtt{hb}^{-1})$$

Put differently, GetRevs tries to create executions with LB cycles only when an LB race is detected.

That said, simply revisiting porf-prior events is not a sound way of generating executions with LB cycles. The problem is that when a write w revisits a porf-earlier read r, the graph that Restrict would create also includes r's po-suffix. This po-suffix *has* to be removed from the graph, since its very existence may depend on the value read by r (e.g., due to control flow). Yet, some events of the po-suffix do need to be re-added in the graph if porf-cyclic executions are to be generated.

WMC resolves this problem with a two-step approach. As a first step, when a write w revisits a porf-earlier read r, WMC removes the porf-prefix of w that is po-after r (Line 8). The writes that are po-after r and are read externally are kept in a *promise set* calculated by Promises (Line 9), which is in turn used during the recursive exploration (Line 10). Promise sets, Π, are sets of pairs consisting of the write that is promised and the execution graph when the promise was issued (the latter is used to constrain the reads within promise certifications).

As a second step, whenever Visit encounters a non-empty promise set, WMC initiates a *certification phase*. During this phase, WMC operates in a restricted mode: all promises of Π have to be fulfilled (i.e., the corresponding writes have to be re-added to the graph), and all non-local explorations (see §2.3) are postponed until the certification phase succeeds.

To that end, WMC modifies GenMC's original algorithm in the following ways.

First, the next function is changed so that when WMC is in a certification phase (i.e., $\Pi \neq \emptyset$), next returns the events of the thread under certification until either all promises have been fulfilled, or all the events of this thread have been added to the graph (in which case next returns $\perp$).

Second, the GetRFs function is modified to constrain the possible reads-from options during the certification phase (Line 3). The modified version of the function calculates the set of local writes that were used to issue promises (Line 4). Reading from these writes is safe, since it cannot lead to bait-and-switch behaviors. All other (non-local) reads-from options will be considered only after the certification procedure has succeeded (see below).

Third, VisitRevisits is changed to behave differently under certification. After Visit removes any promise in Π fulfilled by a (Line 12), VisitRevisits checks whether the certification is over. If the certification is not over yet, no revisits are performed (Line 2). (Analogously to GetRFs, a's revisits will be calculated later once the certification procedure completes successfully.) If the certification is over (i.e., all promises have been fulfilled), then all rf options that were discarded by GetRFs and all revisits that were skipped by VisitRevisits will now be considered (Line 4). To that end, VisitRevisits calculates all non-local options for the reads and writes of the thread that completed the certification (by means of CertRevs; Line 4), and then recursively explores them.

Arguably, the most intricate parts in the changes described above are (1) the calculation of non-local options at the end of a certification, and (2) the calculation of the promise set for a recursive exploration. But, before diving into these parts, let us see an example of WMC in action.

4.2.2 WMC: An Example. Using the modifications above, WMC explores all 4 executions of Fig. 1.

Initially, the exploration remains the same as with GenMC. In graph ⑥, however (and in contrast to GenMC), WMC also considers $\mathtt{R}(y)$ to be revisited by $\mathtt{W}(y,1)$, eventually leading to graph ② below, where $\mathtt{W}(x,1)$ needs to be certified:

$$\begin{tikzcd}
\textcircled{1} & & {[init]} \arrow[dl] \arrow[dr] \arrow[dr, dashed, bend left, "\mathtt{rf}"] & & & \textcircled{2} & & {[init]} \arrow[dl] \arrow[dr] & \\
& \mathtt{R}(x) \arrow[d] & & \mathtt{R}(y) \arrow[d] & \rightsquigarrow \cdots \rightsquigarrow & & \mathtt{R}(x) \arrow[d] & & \mathtt{R}(y) \\
& \mathtt{W}(y,1) & & \mathtt{W}(x,1) \arrow[ull, dashed, "\mathtt{rf}"] & & & \mathtt{W}(y,1) \arrow[urr, dashed] & & {} \arrow[ull, dashed]
\end{tikzcd}$$

Let us now see what happens when Visit proceeds with exploration corresponding to graph ②. Since Π is non-empty, WMC enters a certification phase. The next event to be added is $\mathtt{W}(x,1)$, which fulfills the promised write of the first thread. Since no revisits were skipped (and no rf edges were restricted) during the certification phase, CertRevs returns the empty set, and the execution (corresponding to Fig. 1d) is complete.

The rest of the exploration proceeds in a similar manner. When WMC encounters graph ⑧, it will notice that graph ⑧ contains an LB race, and therefore generate another (duplicate) execution with the weak LB behavior[4]. In general, however, this is not something WMC could have predicted, and thus has no way of avoiding it without doing some extra bookkeeping (see below).

We conclude this example with two remarks.

[4]The exploration is similar to the one presented for ② above, and is thus omitted for brevity.

Blocked Executions. While in the LB-NODEP program above all promises could be fulfilled, this is not the case in general. In the LB+CTRL program below, whenever a read gets revisited in hope that a cyclic execution will be generated (as in execution ② above), its promise cannot be fulfilled because of the control dependency between the read and its subsequent write. Although there can be a fair number of blocked executions in a test case, the overhead that is induced on WMC by such explorations is modest because WMC does not fully explore these blocked executions: it discards them as soon as the certification phase fails.

```
r1 := x              ||  r2 := y                           ||  r2 := z  ||
if r1 = 0 then       ||  if r2 = 0 then   (LB+CTRL)  r1 := x   ||  r3 := y  ||  z := 2    (RLB+W)
  y := 1             ||    x := 1                    y := 1    ||  x := 1   ||
```

Duplicate Executions. One may wonder what does WMC do about duplicate executions in general. For LB-race-free programs, WMC essentially follows GenMC: instead of exploring the executions of a program recursively, WMC uses a stack and some auxiliary structures that allow it to not encounter any duplication at the cost of some extra memory (see [Kokologiannakis et al. 2019]).

For LB-racy programs like LB-NODEP, however, it is not possible to avoid exploring duplicate executions simply by recording the LB cycles that have already been encountered in the exploration. Indeed, it turns out this naive approach of recording LB cycles is not sound, as it may lead to the disposal of valid executions. To see this, consider the RLB+W program above and assume that we record the encountered cycles. Note that RLB+W has two executions with LB cycles: one where $r_2 = 0$, and one where $r_2 = 2$. Assuming that events are added from left to right and that we first encounter the cyclic execution where $r_2 = 0$, when we encounter the second execution where $r_2 = 2$, it will be erroneously disposed, as it involves the exact same cycle as the one with $r_2 = 0$.

A sound way to avoid duplication is to record the whole porf-prefix of such LB cycles. Although this can amount to recording complete execution graphs, it does not induce a significant space overhead in practice because executions with LB cycles are rare.

4.2.3 WMC: Promises and Non-Local Revisits. Finally, let us now return to the last remaining parts of VisitRevisits: the calculation of non-local options at the end of a certification, and the calculation of the promise set Π.

Starting with the calculation of non-local options, when the certification is over, we have to a) calculate alternative rf edges for the reads that read locally during certification, and b) calculate revisits for writes the revisits of which were skipped during certification. As can be seen in Algorithm 2, CertRevs performs exactly these two actions.

Continuing with the calculation of the promise set, given a revisit $\langle w, r\rangle$, the definition of Promises simply returns the writes after po-after r that are read externally, and are porf-before its last parameter S.

The only question that remains the be answered is *what* should be passed as the S argument of Promises (cf. VisitRevisits, Line 9). Clearly, one thing that should be passed as part of S is the revisiting write w indeed, this is the first component passed to Promises. Passing just w, however, is not enough. In fact, there are two more cases we have to take into account.

As an example of the first case, consider the end of a successful certification phase triggered by a revisit $\langle w, r\rangle$, as well as a revisit $\langle w', r'\rangle$ returned by CertRevs at the certification end. Let us further assume that the revisit $\langle w', r'\rangle$ initiates a new certification phase. In such cases, it is inadequate to consider as promises only the writes that are porf-before w'. Since w' is itself porf-before w (and part of the cycle that the revisit $\langle w, r\rangle$ created), we also have to include w in S in order to ensure that w's cycle will be preserved; that is the role of S's second component, Π_c. (Note that revisiting r' from w' without preserving the cycle of $\langle w, r\rangle$ is also possible, but that execution will be obtained in another graph without the $\langle w, r\rangle$ cycle.)

As an example of the second case, consider the program below along with its Weakestmo2-consistent execution where all reads read 1:

$$
z := 1 \;\left\|\; \begin{array}{l} r := z \\ s := x \\ \textbf{if } r = 0 \lor s = 1 \textbf{ then} \\ \quad y := 1 \end{array} \;\right\|\; \begin{array}{l} a := y \\ x := a \end{array}
$$

$$\begin{tikzcd}
 & {[init]} \arrow[dl] \arrow[d] \arrow[dr] & \\
\mathtt{W}(z,1) \arrow[r, dashed] & \mathtt{R}(z,1) \arrow[d] & \mathtt{R}(y,1) \arrow[d] \\
 & \mathtt{R}(x,1) \arrow[d] & \mathtt{W}(x,1) \arrow[l, dashed] \\
 & \mathtt{W}(y,1) \arrow[uur, dashed] &
\end{tikzcd}$$

Perhaps surprisingly, it is impossible to get the above graph if we do not revisit the read of z *after* the LB-cycle between x and y has been created. Indeed, let us assume that WMC executes the program in a left to right manner. When the second thread is executed, $r := z$ can read either 0 or 1, while $s := x$ can only read 0. Given these options, the write $y := 1$ (and, by extension, the LB cycle) will only appear in the exploration where $r := z$ reads 0. In other words, while it is consistent to have both reads of the second thread reading 1, it is impossible to arrive at this scenario after the first read reads 1.

To account for this problem, whenever a certification is complete and an LB cycle is created, we have to (1) revisit reads in the certification thread that are po-before the cycle, and (2) ensure that the cycle will continue to exist after these revisits take place. Luckily, our definition of CertRevs already alleviates the first issue: instead of considering revisits just for reads that participate in the cycle, it also considers revisits of reads that are po-before the cycle. To solve the second issue (i.e., guarantee that the cycle will exist after these revisits take place), we use the InCyc function, which returns all writes in the certification thread that are po-after its argument, and also participate in the cycle[5].

5 EVALUATION

We evaluate WMC by answering the following questions:

§5.1 How often do LB races appear in practice? What overhead is there for detecting them?
§5.2 How well does WMC perform against other tools that handle similar memory models?
§5.3 How does WMC scale in synthetic benchmarks containing many load buffering races?
§5.4 Can we use WMC to automatically remove load buffering races?

To do so, we compare WMC against the following tools.

- GenMC [Kokologiannakis et al. 2019; 2021] is the stateless model checker that we build upon. It supports the RC11 memory model, which does not permit LB behaviors.
- HMC [Kokologiannakis et al. 2020] is an extension of GenMC that verifies programs under IMM [Podkopaev et al. 2019], a weak memory model that allows certain LB behaviors by keeping track of dependencies between instructions and forbidding cycles consisting solely of dependencies and `rf` edges.
- HMC$_{\mathrm{LBRF}}$ is a variant of HMC that we implemented in order to further demonstrate the benefits of LBRF. HMC$_{\mathrm{LBRF}}$ leverages LBRF and starts calculating dependencies only when it finds a load buffering race.
- Nidhugg [Abdulla et al. 2015a; 2016] is a stateless model checker that, among others, supports the POWER memory model by tracking dependencies (similarly to HMC).
- CDSChecker [Norris et al. 2013] verifies programs under an informally defined strengthening of the original C11 model, which forbids thin-air behaviors using a notion of promises.

[5]As a further optimization, InCyc returns the empty set if its argument is reading non-locally. We do not have to consider non-local reads, as such reads will be local in another exploration.

- RMEM [rmem 2009] is a memory model simulator that supports a number of hardware memory models. In our benchmarks, we employ the Promising-Arm model [Pulte et al. 2019] because RMEM usually runs much faster with Promising-Arm than with any of its other supported models that allow LB. As the name suggests, Promising-Arm also uses a notion of promises to induce load-buffering behaviors while forbidding thin-air behaviors.

In summary, we observe that LB races are fairly rare in our set of non-synthetic benchmarks and that exploiting LBRF and CL makes WMC a very effective model checker that has a very small average overhead over GenMC and outperforms the other tools.

Experimental Setup. We conducted all experiments on a Dell PowerEdge M620 blade system, with two Intel Xeon E5-2667 v2 CPUs (8 cores @ 3.3 GHz) and 256GB of RAM, running a custom Debian-based distribution. We used LLVM 7 for HMC (v0.5), GenMC (v0.5), and Nidhugg (v0.3), commit #da671f7 for CDSChecker, and commit #85c8130 for RMEM (v0.1). All reported times are in seconds, unless explicitly noted otherwise. We set the timeout limit to 30 minutes.

5.1 Load Buffering Races and WMC's Overhead

To measure the LB race detection overhead, we conducted two case studies.

In the first case study, we took the lock implementations used by Oberhauser et al. [2021] (13 in total), 6 queue implementations used by GenMC, and 10 data-structure benchmarks used by Ou et al. [2018][6]. Running WMC on these benchmarks confirmed our expectation that realistic implementations rarely contain LB races: out of 29 implementations, we found LB races only in 2. One of them was due to the porting of a non-C11-compliant queue to C11, while the other was an intentional race part of a lock implementation (`musl_lock`), that could not lead to LB behaviors.

In the second case study, we took the entire GenMC benchmark suite (241 tests), which is a combination of small litmus tests and larger concurrent programs. We split these programs into two categories: those that have LB races (28 tests, mostly litmus tests) and those that do not (213 tests). The results are shown in Table 1.

Table 1. Overhead of LBRF on GenMC benchmarks

	GenMC	WMC	HMC	HMC_{LBRF}
LB-racy	0.53	0.55	0.71	0.76
LB-race-free	111.82	134.09	235.85	155.06

As far as WMC is concerned, detecting LB races imposes roughly 25% overhead over GenMC, which is substantially lower than the overhead of calculating dependencies in HMC, especially when no LB races are present.

As far as HMC_{LBRF} is concerned, observe that HMC_{LBRF} significantly improves the running time of HMC on benchmarks without LB races, as it effectively runs WMC, and imposes a negligible overhead with respect to HMC on benchmarks with LB races. The reason for the latter that it typically detects LB races very quickly, in a fraction of the time needed for verifying the program.

In addition, also observe that HMC_{LBRF} performs really close to GenMC and WMC in benchmarks without LB races. The reason for the additional slowdown compared to GenMC and WMC is that HMC_{LBRF} has to maintain some extra data structures which will be necessary if it has to fall back to dependency tracking.

5.2 Comparison with Other Model Checkers

In order to compare WMC and HMC_{LBRF} with other model checkers, we use both synthetic benchmarks and more realistic data structure benchmarks from the literature.

[6]We could not port all 43 of their benchmarks as they are written in C++, only a subset of which is supported by GenMC.

Table 2. Synthetic benchmarks adapted from SV-COMP [2019]

	GenMC	HMC	HMC$_{\text{LBRF}}$	Nidhugg	rmem	CDSChecker	WMC
reorder(2)	0.06	0.11	0.07	1.37	77.10	0.04	0.06
singleton	0.01	0.01	0.01	0.12	96.21	0.00	0.01
fib_bench	13.09	25.98	26.68	237.73	🕒	🕒	37.66
szymanski(1)	0.05	0.07	0.08	0.13	5.56	0.28	0.06
szymanski(2)	246.23	399.79	402.34	3.74	529.85	1688.60	321.23
szymanski(3)	🕒	🕒	🕒	170.88	🕒	🕒	🕒
peterson(10)	0.03	0.06	0.07	0.65	🕒	0.25	0.14
peterson(20)	0.12	0.31	0.30	3.81	🕒	1.36	1.09
peterson(30)	0.32	0.84	0.85	13.54	🕒	3.86	4.43
dekker	0.01	0.02	0.02	0.14	30.09	0.23	0.02
sigma	141.56	352.86	168.94		🕒	🕒	157.91
indexer(12)	0.02	0.02	0.02		🕒	0.90	0.02
indexer(13)	0.05	0.07	0.07		🕒	116.71	0.05
indexer(14)	0.30	0.48	0.50		🕒	🕒	0.34
indexer(15)	2.45	4.03	4.07		🕒	🕒	2.85

Synthetic Benchmarks. We extracted synthetic benchmarks from SV-COMP [2019] (`pthread` and `pthread-atomic` categories) and rewrote them to utilize weakly-ordered atomic accesses so as to contain LB races (see Table 2).

In a nutshell, throughout Table 2, rmem and Nidhugg are generally much slower than all other tools: rmem maintains a total order of stores across all different memory locations, while Nidhugg requires expensive consistency checks at each program step. We also note that Nidhugg does not support RMW instructions and thus some entries in Table 2 are blank.

In addition, as also noted in § 5.1, HMC is approximately 2x slower than GenMC; HMC$_{\text{LBRF}}$, however, does not always perform that much worse compared to GenMC. More specifically, when LB races are present (e.g., for `fib_bench`, `szymanski` and `peterson`), HMC$_{\text{LBRF}}$ performs similarly to HMC. When LB races are not present though (e.g., for `sigma`), HMC$_{\text{LBRF}}$ outperforms HMC and performs similarly to GenMC, providing us a first testament to LBRF's usefulness.

Let us now move to a more detailed comparison between the different tools.

Starting from the upper part of Table 2, we observe that CDSChecker is faster than all other tools for the first two benchmarks. This is attributed to two factors: (1) CDSChecker operates on binaries thus avoiding the interpretation overhead that GenMC, HMC, and Nidhugg have to face, and (2) CDSChecker utilizes a coarser equivalence partitioning in its DPOR that does not totally order the stores of each memory location; that way, CDSChecker explores fewer executions than the other tools. Although GenMC, HMC, and WMC can also operate under a similar partitioning, we chose to not use it, so that WMC better reflects the model presented in §3.

For the next three benchmarks, however, the situation is reversed: CDSChecker performs much worse compared to GenMC, and HMC, despite its coarser equivalence partitioning. In fact, for `szymanski`, both GenMC and HMC are faster than CDSChecker, even though they explore two orders of magnitude more executions than CDSChecker. This big disparity in running times is due to CDSChecker exploring a large number of infeasible executions, caused by unfulfilled promises.

Nevertheless, WMC's performance is not on par with that of GenMC and HMC for the same three benchmarks, as one might have hoped: WMC is slower than GenMC in all three benchmarks, while it manages to outperform HMC only for `szymanski`. For `peterson` and `szymanski` WMC's performance is not attributed to scalability limitations, but rather to Weakestmo2: for these benchmarks the model allows for more executions compared to the models of the other tools. Still, for `szymanski`, WMC is faster than HMC/HMC$_{\text{LBRF}}$ despite the fact that it explores approximately 18% more executions, as calculating dependencies proves to be more expensive. For `fib_bench`, the slowdown WMC experiences is due to the many promises that remain unfulfilled: CDSChecker

Table 3. Benchmarks adapted from Norris et al. [2013]

	GenMC	HMC	HMC$_{LBRF}$	CDSChecker	WMC
barrier-bnd	1.29	2.67	1.60	4.13	1.34
mpmc-queue-bnd	1.87	4.18	2.14	19.78	2.09
treiber-stack-bnd	0.56	1.28	0.65	🕒	0.58
linuxrwlocks-bnd	0.29	0.57	0.58	🕒	0.46
seqlock-bnd	45.40	109.24	111.21	🕒	47.78

suffers from the same problem in `fib_bench`, but does not even manage to terminate within the time limit, as it explores all possible promises, and not just those dictated by `LBRF`.

Finally, we mention in passing that Nidhugg is faster than GenMC, HMC and WMC in `szymanski` due to the way the latter tools handle SC fences. These tools treat SC accesses and fences as release/acquire, as part of an optimization. Thus, they explore 5 orders of magnitude more executions for `szymanski` compared to Nidhugg, resulting in worse performance for this benchmark.

Moving on to the lower part of Table 2, WMC outperforms CDSChecker by a large margin. Take `indexer`, for instance: CDSChecker explores 4 orders of magnitude more infeasible executions than consistent executions, which makes it much slower compared to GenMC, HMC and WMC.

We conclude the discussion for this table with an observation. While WMC was outperformed by HMC and HMC$_{LBRF}$ in a few cases, benchmarks where that happens employ little to no synchronization: this is extremely uncommon for realistic benchmarks, since using relaxed accesses with no other synchronization gives no guarantees. On the other hand, in cases where synchronization is purposefully employed (e.g., `dekker`), or where there are no LB races (e.g., `indexer`), WMC greatly outperforms all other tools that handle similar memory models.

Data Structure Benchmarks. We next consider some more realistic benchmarks (cf. Table 3) from Norris et al. [2013], whose loops have been manually unrolled to ensure fairness across tools. We exclude Nidhugg from the comparison because it does not support RMW instructions under POWER, as well as rmem because it is difficult to encode these benchmarks in its input language.

The observations here are similar to the ones we made for Table 2. CDSChecker performs worse than all other tools due to exploring too many infeasible executions. For the first three benchmarks where there are no LB races, WMC outperforms all other tools operating under similar memory models. In addition, HMC$_{LBRF}$ outperforms HMC, as it operates under RC11. However, even in the last two benchmarks where there are LB races, WMC outperforms all other tools, as it avoids the overhead of tracking dependencies. In the case of `linuxrwlocks` specifically, WMC does have some overhead due to LB races and promises that remain unfulfilled, but it still remains very competitive. Again note that the overhead of detecting LB races by HMC$_{LBRF}$ over HMC is negligible in these examples because an LB race is found in the first few executions.

5.3 Load-Buffering Benchmarks

We next evaluate WMC's performance on synthetic test cases with many LB races, and hence potentially many duplicate executions (cf. Table 4). Although such patterns appear rarely in non-adversarial programs, it is pedagogical to examine how WMC performs in such cases.

Test cases `LB+ctrl(N)` and `LB+data(N)` are similar to program LB from §1, except that in these tests the `porf` cycle spans N threads. Also, `LB+ctrl(N)` has control dependencies between instructions in place of data dependencies as in `LB+data(N)` and LB. Test case `LB-nodep(N)` resembles litmus test LB-NODEP, but, similarly, its `porf` cycle spans N threads. Finally, `LB-pairs(N)` contains N/2 independent pairs of threads with each pair constituting a load buffering pattern with no dependencies.

We used two versions of WMC (cf. first two distinct columns of Table 4): one where duplicate executions are fully explored, and one where executions that will lead to duplicates are blocked as soon as they are detected, using the technique of §4.

WMC explores the same number of unique executions for all benchmarks of Table 4 except for `LB+ctrl`; the latter has control dependencies that preclude LB executions. Except for `LB-pairs`, the percentage of duplicate/blocked executions remains very low or zero. For `LB-pairs`, memorizing cyclic executions that have been explored and pruning unnecessary exploration prevents the blocked executions outgrowing the number of consistent executions and thus dominating verification time.

Table 4. Load buffering benchmarks

	# Execs	Dupls	Blocked
LB+ctrl(10)	11	0	0
LB+ctrl(12)	13	0	0
LB+ctrl(14)	15	0	0
LB+data(10)	1024	0	0
LB+data(12)	4096	0	0
LB+data(14)	16 384	0	0
LB-nodep(10)	1024	0.9%	0.9%
LB-nodep(12)	4096	0.3%	0.3%
LB-nodep(14)	16 384	0.1%	0.1%
LB-pairs(10)	1024	205.2%	33.3%
LB-pairs(12)	4096	281.5%	33.3%
LB-pairs(14)	16 384	376.8%	33.3%

5.4 Automatically Fixing Load-Buffering Races

Finally, we investigate how easily LB races can be eliminated. For that, we constructed a script that tries to automatically repair a test case by strengthening the access mode of some instructions in the race's `porf` path. The script does that incrementally: when an LB race is detected by WMC, the script modifies the test's source code by strengthening the access mode of two instructions constituting an `rf` edge in the race's `porf` path, and then runs WMC again until no race exists.

We ran our script on all 30 benchmarks with LB races from §5.1. In most cases (21/30), our script eliminated the LB races with a single iteration. Four tests needed two iterations to be repaired, while one test (`szymanski`) required 10 iterations. The reason is that `szymanski` employs very little synchronization in the form of SC fences. As such, it contains many LB races that need to be repaired. Finally, there were 4 benchmarks which the script was unable to repair: since our script performs syntactic transformations to the test's source code, it cannot repair tests that use custom primitives for shared memory accesses.

6 RELATED WORK AND CONCLUSION

There has been a lot of work on developing weak memory models for Java/C/C++ that resolve the "out-of-thin-air" (OOTA) problem [Chakraborty et al. 2019; Jagadeesan et al. 2020; Jeffrey et al. 2016; Kang et al. 2017; Lee et al. 2020; Manson et al. 2005; Paviotti et al. 2020; Pichon-Pharabod et al. 2016] with more recent solutions typically criticizing the earlier ones because of their outcomes on certain litmus tests or the authors' stylistic preferences. Given the lack of an agreed formal criterion for classifying OOTA behaviors, load buffering race freedom and certification locality can be seen as more rigorous appraisals of weak memory models. That said, although LBRF is defined in a model-independent fashion, CL is much more tied to the certification mechanism that is present in the Promising [Kang et al. 2017; Lee et al. 2020] and Weakestmo [Chakraborty et al. 2019] models, and may not be easy to apply to multi-execution memory models with a very different definitional structure. Somewhat surprisingly, both Promising and Weakestmo violate both LBRF and CL; yet, as we have shown, Weakestmo can be adapted to satisfy these properties, while maintaining correctness of the compilation schemes to hardware architectures.

Recently, Jagadeesan et al. [2020] developed a very interesting multi-execution memory model that allows LB but forbids bait-and-switch behaviors, as well as a temporal logic that can be used to dismiss the dubious outcome of LBF-CEX. We expect that their model satisfies LBRF and probably also CL, but have not yet investigated a proof of these conjectures.

In terms of program verification under weak memory models, there is a significant body of work for porf-acyclic models, such as TSO and RC11. These range from program logics for manual verification (e.g., [Doko et al. 2016; 2017; Kaiser et al. 2017; Ridge 2010; Sieczkowski et al. 2015; Turon et al. 2014]) to model checking tools for safety verification (e.g., [Abdulla et al. 2015a; 2018; Barnat et al. 2013; Demsky et al. 2015; Huang et al. 2016; Kokologiannakis et al. 2017; 2019]) and fence insertion tools for enforcing robustness (e.g., [Abdulla et al. 2015b; Bouajjani et al. 2013]). As discussed in §2.2, LBRF provides a simple mechanism for programmers to use these results on weaker memory models at the cost of a few extra fences. Similarly, there are a number of papers that can handle dependency-tracking models, such as ARM and POWER (e.g., [Abdulla et al. 2016; Alglave et al. 2017; 2013; Kokologiannakis et al. 2020; Pulte et al. 2019]).

By contrast, there is hardly any work on verifying concurrent programs on multi-execution memory models: Svendsen et al. [2018] develop a program logic for Promising, which is considerably simpler than corresponding logics for TSO and RC11 [Doko et al. 2016; 2017; Kaiser et al. 2017; Sieczkowski et al. 2015; Turon et al. 2014], and CDSChecker [Norris et al. 2013] is a model checker for an earlier adaptation of the C11 model with promises similar to those in Promising. As discussed, the reason for this lack of work is that multi-execution models are substantially more complicated and not very amenable to automated formal verification.

The notions we introduced in this work—load buffering race freedom and certification locality—are first steps towards enabling formal verification for such multi-execution models, and indeed have been instrumental in the design of WMC. We hope that they will lead to further exploration in this space, and that they will also prove useful for program logics.

ACKNOWLEDGMENTS

We would like to thank Soham Chakraborty and Anton Podkopaev for their insights on Weakestmo and IMM memory models, and the anonymous reviewers for their valuable feedback. This work was supported by a European Research Council (ERC) Consolidator Grant for the project "PERSIST" under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

REFERENCES

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

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas (2014). "Optimal dynamic partial order reduction." In: *POPL 2014*. New York, NY, USA: ACM, pp. 373–384. DOI: 10.1145/2535838.2535845.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Adwait Godbole, S. Krishna, and Viktor Vafeiadis (2021). "The Decidability of Verification under PS 2.0." In: *ESOP 2021*. Ed. by Nobuko Yoshida. Cham: Springer International Publishing, pp. 1–29.

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

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

Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Ngo Tuan Phong (2015b). "The best of both worlds: Trading efficiency and optimality in fence insertion for TSO." In: *ESOP 2015*. Vol. 9032. LNCS. Springer, pp. 308–332. DOI: 10.1007/978-3-662-46669-8_13.

Sarita V. Adve and Kourosh Gharachorloo (Dec. 1996). "Shared memory consistency models: A tutorial." In: *IEEE Comput.* 29.12, pp. 66–76.

Jade Alglave and Patrick Cousot (2017). "Ogre and Pythia: an invariance proof method for weak consistency models." In: *POPL 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, pp. 3–18. URL: http://dl.acm.org/citation.cfm?id=3009883.

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

Jade Alglave, Luc Maranget, and Michael Tautschnig (July 2014). “Herding cats: Modelling, simulation, testing, and data mining for weak memory.” In: *ACM Trans. Program. Lang. Syst.* 36.2, 7:1–7:74. DOI: 10.1145/2627752.

J. Barnat, L. Brim, and V. Havel (July 2013). “LTL model checking of parallel programs with under-approximated TSO memory model.” In: *ACSD 2013*, pp. 51–59. DOI: 10.1109/ACSD.2013.8.

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell (2015). “The problem of programming language concurrency semantics.” In: *ESOP 2015*. Vol. 9032. LNCS. Berlin, Heidelberg: Springer, pp. 283–307. URL: http://dx.doi.org/10.1007/978-3-662-46669-8_12.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber (2011). “Mathematizing C++ concurrency.” In: *POPL 2011*. Austin, Texas, USA: ACM, pp. 55–66. DOI: 10.1145/1926385.1926394.

Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer (2013). “Checking and enforcing robustness against TSO.” In: *ESOP 2013*. Vol. 7792. LNCS. Springer, pp. 533–553.

Soham Chakraborty and Viktor Vafeiadis (Jan. 2019). “Grounding thin-air reads with event structures.” In: *Proc. ACM Program. Lang.* 3.POPL, 70:1–70:28. DOI: 10.1145/3290383.

Brian Demsky and Patrick Lam (2015). “SATCheck: SAT-directed stateless model checking for SC and TSO.” In: *OOPSLA 2015*. Pittsburgh, PA, USA: ACM, pp. 20–36. DOI: 10.1145/2814270.2814297.

Marko Doko and Viktor Vafeiadis (2016). “A Program Logic for C11 Memory Fences.” In: *VMCAI 2016*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. LNCS. Springer, pp. 413–430. DOI: 10.1007/978-3-662-49122-5_20.

Marko Doko and Viktor Vafeiadis (2017). “Tackling Real-Life Relaxed Concurrency with FSL++.” In: *ESOP 2017*. Ed. by Hongseok Yang. Vol. 10201. LNCS. Springer, pp. 448–475. DOI: 10.1007/978-3-662-54434-1_17.

Cormac Flanagan and Patrice Godefroid (2005). “Dynamic partial-order reduction for model checking software.” In: *POPL 2005*. New York, NY, USA: ACM, pp. 110–121. DOI: 10.1145/1040305.1040315.

Shiyou Huang and Jeff Huang (2016). “Maximal Causality Reduction for TSO and PSO.” In: *CONF_OOPSLA 2016*. New York, NY, USA: ACM, pp. 447–461. DOI: 10.1145/2983990.2984025.

Radha Jagadeesan, Alan Jeffrey, and James Riely (Nov. 2020). “Pomsets with preconditions: A simple model of relaxed memory.” In: *JNL_PACMPL* 4.OOPSLA. DOI: 10.1145/3428262.

Alan Jeffrey and James Riely (2016). “On thin air reads: Towards an event structures model of relaxed memory.” In: *CONF_LICS 2016*. New York, NY, USA: ACM, pp. 759–767. DOI: 10.1145/2933575.2934536.

Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev (2022). “The leaky semicolon: Compositional semantic dependencies for relaxed-memory concurrency.” In: *JNL_PACMPL* 6.POPL, pp. 1–30. DOI: 10.1145/3498716.

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

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer (2017). “A promising semantics for relaxed-memory concurrency.” In: *CONF_POPL 2017*. Paris, France: ACM, pp. 175–189. DOI: 10.1145/3009837.3009850.

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

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis (2019). “Model checking for weakly consistent libraries.” In: *PLDI 2019*. New York, NY, USA: ACM. DOI: 10.1145/3314221.3314609.

Michalis Kokologiannakis and Viktor Vafeiadis (2020). “HMC: Model checking for hardware memory models.” In: *ASPLOS 2020*. ASPLOS ’20. Lausanne, Switzerland: ACM, pp. 1157–1171. DOI: 10.1145/3373376.3378480.

Michalis Kokologiannakis and Viktor Vafeiadis (2021). “GenMC: A model checker for weak memory models.” In: *CAV 2021*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. LNCS. Springer, pp. 427–440. DOI: 10.1007/978-3-030-81685-8_20.

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer (2017). “Repairing sequential consistency in C/C++11.” In: *PLDI 2017*. Barcelona, Spain: ACM, pp. 618–632. DOI: 10.1145/3062341.3062352.

Leslie Lamport (Sept. 1979). “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs.” In: *IEEE Trans. Computers* 28.9, pp. 690–691. DOI: 10.1109/TC.1979.1675439.

Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis (2020). “Promising 2.0: Global optimizations in relaxed memory concurrency.” In: *PLDI 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, pp. 362–376. DOI: 10.1145/3385412.3386010.

Jeremy Manson, William Pugh, and Sarita V. Adve (2005). “The Java memory model.” In: *POPL 2005*. ACM, pp. 378–391. DOI: 10.1145/1040305.1040336.

Evgenii Moiseenko, Michalis Kokologiannakis, and Viktor Vafeiadis (2022). *Model Checking for a Multi-Execution Memory Model (Supplementary Material)*. URL: https://plv.mpi-sws.org/wmc/.

Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis (2020). “Reconciling Event Structures with Modern Multiprocessors.” In: *ECOOP 2020*. Vol. 166. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 5:1–5:26. DOI: 10.4230/LIPIcs.ECOOP.2020.5.

Brian Norris and Brian Demsky (2013). “CDSChecker: Checking concurrent data structures written with C/C++ atomics.” In: *OOPSLA 2013*. ACM, pp. 131–150. DOI: 10.1145/2509136.2509514.

Jonas Oberhauser et al. (2021). “VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models.” In: *ASPLOS 2021*. Virtual, USA: ACM, pp. 530–545. DOI: 10.1145/3445814.3446748.

Peizhao Ou and Brian Demsky (Oct. 2018). “Towards Understanding the Costs of Avoiding Out-of-Thin-Air Results.” In: *Proc. ACM Program. Lang.* 2.OOPSLA. DOI: 10.1145/3276506.

Scott Owens, Susmit Sarkar, and Peter Sewell (2009). “A better x86 memory model: x86-TSO.” In: *TPHOLs 2009*. Munich, Germany: Springer, pp. 391–407. DOI: 10.1007/978-3-642-03359-9_27.

Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty (2020). “Modular relaxed dependencies in weak memory concurrency.” In: *ESOP 2020*. Ed. by Peter Müller. Vol. 12075. LNCS. Springer, pp. 599–625. DOI: 10.1007/978-3-030-44914-8_22.

Jean Pichon-Pharabod and Peter Sewell (2016). “A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions.” In: *POPL 2016*. St. Petersburg, FL, USA: ACM, pp. 622–633. DOI: 10.1145/2837614.2837616.

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

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

Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur (2019). “Promising-ARM/RISC-V: A simpler and faster operational concurrency model.” In: *PLDI 2019*. Phoenix, AZ, USA: ACM, pp. 1–15. DOI: 10.1145/3314221.3314624.

Tom Ridge (2010). “A rely-guarantee proof system for x86-TSO.” In: *VSTTE 2010*. Vol. 6217. LNCS. Springer, pp. 55–70.

rmem (2009). *rmem: Executable concurrency models for ARMv8, RISC-V, Power, and x86*. URL: https://github.com/rems-project/rmem (visited on Aug. 24, 2019).

Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod (2015). “A separation logic for fictional sequential consistency.” In: *ESOP 2015*. Vol. 9032. LNCS. Berlin, Heidelberg: Springer, pp. 736–761.

SV-COMP (2019). *Competition on Software Verification (SV-COMP)*. URL: https://sv-comp.sosy-lab.org/2019/ (visited on Mar. 27, 2019).

Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis (2018). “A separation logic for a promising semantics.” In: *ESOP 2018*. Ed. by Amal Ahmed. Vol. 10801. LNCS. Springer, pp. 357–384. DOI: 10.1007/978-3-319-89884-1_13.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer (2014). “GPS: Navigating weak memory with ghosts, protocols, and separation.” In: *OOPSLA 2014*. ACM, pp. 691–707. DOI: 10.1145/2660193.2660243.