Model Checking on Multi-execution Memory Models

EVGENII MOISEENKO, JetBrains Research
MICHALIS KOKOLOGIANNAKIS, MPI-SWS, Germany
VIKTOR VAIFEIADIS, 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 surprisingly 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:

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.)

\[
\begin{align*}
  r_1 &:= x \text{ \# reads } 1 \\
  y &:= 1 \\
  r_2 &:= y \text{ \# reads } 1 \\
  x &:= r_2
\end{align*}
\]

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 \( x = 1 \).
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 [Kokologian-nakis 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 \( \text{porf} = (\text{po} \cup \text{rf})^+ \) 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 \( \text{hb} = \text{po} \).

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 \( \text{porf} \) 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 \( \text{porf} \) 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 \( \text{porf} \) 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
races. We say that two events (of different threads) are concurrent if they are not ordered by 
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 \( \text{DRF}_M \)) 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 \( \text{DRF}_{SC} \) 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 pos 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 rpo; (rf \ po) ; poř path from \( r \) to \( w \) (i.e., a poř-path starting with a reorderable edge).

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

Existence of a poř-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 poř-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 \( \text{LBRF}_{M'} \)) 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 poř-cycles, such as RC11. Similar to DRF, \( \text{LBRF}_{RC11} \) 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.
under RC11 (e.g., iGPS, Herd, RCMC, GenMC, etc). Moreover, LB races can easily be removed by making the \texttt{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 LBRF\textsubscript{RC11} because it allows out-of-thin-air outcomes and does not even satisfy DRF\textsubscript{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\textsubscript{wrt} their strengthenings with \texttt{porf}-acyclicity. The additional behaviors they allow over their strengthenings are \texttt{porf}-cycles with at least one \texttt{po} edge from a load to a store being reorderable. By changing the \texttt{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 LBRF\textsubscript{RC11}. *(See Appendix 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 LBRF\textsubscript{RC11}. To see this, consider the LBF-CEX program along with its executions shown in Fig. 2. Although none of its RC11-consistent executions (\textbf{A}, \textbf{B}, \textbf{C}, \textbf{D}) contains an LB race, both Promising and Weakestmo allow the additional execution \textbf{E} where \( r \) gets the value 1. This execution can arise in the following manner. First, thread 2 promises the \( W(y, 1) \) 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.

\[
\begin{align*}
x &:= 1 \\
r_1 &:= x \quad \text{# reads 1} \\
y &:= r_1 \\
r_2 &:= y \quad \text{# reads 1} \\
x &:= r_2
\end{align*}
\]

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 \textbf{LB+LOOP} program below and the associated execution graph that reads an arbitrary natural number \( n \) into \( r_1 \) and \( r_2 \).

\[
\begin{align*}
r_1 &:= x \\
\text{for } i &:= 0 \text{ to } r_1 \\
y &:= i + 1 \\
r_2 &:= y \\
\text{for } j &:= 1 \text{ to } r_2 \\
x &:= j
\end{align*}
\]

\[
\begin{align*}
\text{init} \\
R(x, 1) &\rightarrow W(y, 1) &W(y, 2) &\rightarrow \cdots &\rightarrow W(y, n) &W(y, n + 1) \\
&\rightarrow R(y, 1) &\rightarrow W(x, 1) &\rightarrow W(x, 2) &\rightarrow \cdots &\rightarrow W(x, n)
\end{align*}
\]

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 \( y := 1 \) (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 Appendix 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 LBRFRC11 and DRFSC 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 \(R(x, 0)\) and \(W(x, 1)\). Thus, WMC creates the execution of Fig. 1d by promising the write \(W(y, 1)\) in Thread 1 (so that \(R(y, 1)\) can read from it), and then by making \(R(x, 0)\) read from \(W(x, 1)\). 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 \(W(y, 1)\) in Thread 2.

```plaintext
z := 1
if x then
  r := z
  y := r
else
  y := 1
if y then
  x := 1
W(z, 1) [init] R(x, 0) R(y, 1) (LB+Rdep)
W(y, 1) W(x, 1)
```

Proc. ACM Program. Lang., Vol. 6, No. OOPSLA2, Article 152. Publication date: October 2022.
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 \( W(z, 1) \). 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 \( W(y, 1) \) 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 \( W(z, 1) \) has been added to the graph or not, WMC will only consider the read \( R(z, 0) \), and therefore conclude that the promise \( W(y, 1) \) 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 HMC\(_{LBRF} \), a version of HMC that leverages LBRF. Similarly to WMC, HMC\(_{LBRF} \) starts by enumerating RC11-consistent executions, and falls back to dependency tracking only upon detecting an LB race. As we show in §5, HMC\(_{LBRF} \) 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 Appendix 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 \( \langle \text{id}, \text{tid}, \text{lab} \rangle \) where \( \text{id} \in \mathbb{N} \) is a unique identifier for the event, \( \text{tid} \in \mathbb{N}_\perp \) identifies the thread to which the event belongs (\( \perp \) for initialization events), and \( \text{lab} \) is a label of the form: (1) \( \text{R}(x, v) \) for a read of \( v \in \text{Val} \) from \( x \in \text{Loc} \); (2) \( \text{W}(x, v) \) for a write of \( v \in \text{Val} \) to \( x \in \text{Loc} \); (3) \( \text{F} \) for a fence.

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

Further, given a relation \( r \), we use \( \text{dom}(r) \) and \( \text{rng}(r) \) to denote its domain and codomain, while \( r^\circ, r^+ \), and \( r^* \) denote its reflexive, transitive, and reflexive-transitive closures, respectively. We write
When sufficient to just take a conflict-free subset of events. For example, taking the single event \( \{e_2\} \) can be derived. All reads have a corresponding write that justifies them, i.e., we have, \( \text{init} \rightarrow e_1 \rightarrow e_2 \). For the inverse of \( r \), \([A] \) for the identity relation \( \{(a, a) \mid a \in A\} \), \( r_1 \circ r_2 \) 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 \( =_f \triangleq \{(a, b) \mid f(a) = f(b) \neq \perp\} \).

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

- \( E \subseteq \) Event is a set of events with unique identifiers that includes initialization events, writing 0 to each memory location used by the program.
- \( \text{po} \subseteq \mathbb{E} \times \mathbb{E} \) 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, \( \text{po} \) does not have to be total among the events of the same thread. Events of the same thread unordered by \( \text{po} \) are said to be in conflict: \( \text{cf} \triangleq =_{\text{tid}} \setminus (\text{po} \cup \text{po}^{-1}) \). Two events are in immediate conflict if they are in conflict but their predecessors are not: \( \text{cf}_{\text{imm}} \triangleq \text{cf} \setminus (\text{po} \cup \text{po}^{-1} \cup \text{cf}) \).
- \( \text{jf} \subseteq [\mathbb{W}] ; (=_\text{loc} \cap =_{\text{val}}) \setminus \{R\} \) is the justified from relation, which maps each read event to the write event that justifies it. We assume that \( \text{jf} \) is functional and complete: for each read event there exists a unique write event that justifies it.
- \( \text{ew} \subseteq [\mathbb{W}] ; (\text{cf} \cap =_{\text{loc}} \cap =_{\text{val}}) \setminus \{\mathbb{W}\} \) is the equal-writes relation which relates conflicting writes that are considered equal. Its reflexive-transitive closure \( \text{ew}^* \) is an equivalence relation on write events.
- \( \text{co} \subseteq [\mathbb{W}] ; (=_\text{loc} \setminus \text{ew}^*) \setminus \{W\} \) 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 \( \text{LB} \). We can see, for example, that the events \( e_{11} \rightarrow e_{21} \) 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 \( e_{12} \rightarrow e_{22} \) is shown on the right. Note that we only depict immediate conflict edges. For example, we draw \( e_{11} \leftrightarrow e_{12} \), but not \( e_{11} \leftrightarrow e_{22} \), as the latter can be derived. All reads have a corresponding write that justifies them, i.e., we have, \( \text{init} \rightarrow e_{11} \rightarrow e_{21} \rightarrow e_{12} \rightarrow e_{22} \). The two writes to location \( y \) are considered equal \( e_{21} \leftrightarrow e_{22} \).

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

**Proposition 3.4.** For an execution graph \( G \), we have:

- \( G.\text{po} \) is total on the events of a given thread;
- \( G.\text{ew} \) is the empty relation;
- \( G.\text{co} \) 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 \( \{e_{12}\} \)
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 \( e_1^2 \) 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 \( e_1^{12} \). To achieve that, we define the derived \textit{reads-from} relation \( rf \triangleq ( \text{ew}^* \cap \text{rf}) \setminus \text{cf} \) by extending the \( jf \) relation to \( \text{ew}^* \) equivalence classes.

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

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

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

**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 \( \text{co} \)-maximal write to \( x \) in \( G \).

For example, in Fig. 3 the set of events marked by \( \bigcirc \) 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.

- \( \text{hb} \triangleq (\text{po} \cup \text{sw})^+ \) — \textit{Happens-before} is the transitive closure of the program order and \( \text{synchronizes-with} \) relations. The latter connects synchronized events. For example, it connects two fences if there exists a \( \text{po} ; rf ; \text{po} \) path between them.
- \( \text{ecf} \triangleq (\text{hb}^{-1})^* \cap \text{cf} \) — \textit{Extended conflict} propagates the conflict relation along \( \text{hb} \).
- \( \text{eco} \triangleq (\text{co} \cup \text{rf} \cup \text{rf}^{-1} \cup \text{co})^* \) — \textit{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.
- \( \text{jo} \triangleq (\text{jf} \setminus \text{po}) \cup (\text{po} \cup \text{jf})^* \) — \textit{justification order}. The \( \text{jo} \) 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 \textit{justification set} of the event \( e \).
- A write event \( w \) is a \textit{promise} w.r.t. an event \( e \) if \( (w, e) \in \text{cf} \cap \text{jo} \). Additionally:
  - if \( (w, e) \in \text{ew}^* ; \text{po}^2 \) then \( w \) is a \textit{certified promise};
  - if \( (w, e) \notin \text{ew}^* ; \text{po}^2 \) then \( w \) is a \textit{pending promise}.

Consider Fig. 3 again. Write event \( e_1^{12} \) 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.

- \( \text{ecf} \) is irreflexive. (NON-CONTRADICTORY)
- \( \text{jo} \cap \text{ecf} = \emptyset \) (WELL-JUSTIFIED)
- \( \text{po} \cup \text{jf} \) is acyclic (NO-THIN-AIR)
- \( \text{hb} ; \text{eco}^2 \) is irreflexive. (COHERENT)
- \( [F] ; \text{po} ; \text{ew}^+ \subseteq \text{po} \) (WELL-FENCED)
- \( \text{cf} \cap \text{jo} \subseteq \text{ew}^* ; (\text{po} \cup \text{po}^{-1})^* \) (CERTIFIED)
- \( \text{ew} \subseteq (\text{cf} \cap (\text{jo} \cup \text{jo}^{-1}))^+ \) (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 LBRF\(_{RC11}\) 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 LBRF\(_{RC11}\).

In order to get the cyclic execution of LBF-CEX, Thread 2 first issues the promise \(e_{22}\). Through Thread 3, this promise justifies another branch of Thread 2, namely \(e_{21} \rightarrow e_{21} \rightarrow e_{31}\), and can be certified thanks to the equivalent write \(e_{31}\). 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, \(e_1\) externally justifies \(e_{11}\) and \(e_2\) externally justifies \(e_{21}\). 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:

\[
\begin{align*}
\bullet \ (jf \setminus (jf^2; hb)) &; po ; ew \subseteq jf ; (po \cup \lpapat) \\
\text{(NO-BAIT-AND-SWITCH)}
\end{align*}
\]

where \(\lpapat \triangleq cf_{imm}; [rng(jf \cap (jf^2; hb))]\); po denotes the load buffering pattern.

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

\(^1\)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 §C) filters out executions that violate atomicity of read-modify-write operations or sequential consistency of sc accesses.
The \textsc{no-bait-and-switch} axiom requires that external justifications of one branch either also justify some read in the conflicting branch \((j\bar{f}; po)\) or be the very reason why the branch was created \((j\bar{f} : lbpat)\): 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 \textsc{Weakestmo2}-consistent because event \(e_{21}^3\) externally justifies \(e_{21}^2\), but does not justify any event \(po\)-before \(e_{22}^2\). In contrast, the event structure of Fig. 3 is \textsc{Weakestmo2}-consistent. The only relevant external justification is that of \(e_{11}^1\) by \(e_{21}^2\), which is allowed because \(e_{12}^1\) is in immediate conflict with \(e_{11}^1\), which is \(po\)-before \(e_{21}^2\).

### 3.3 Load Buffering and Data Race Freedom

We recall the definition of \(RC11\)-consistency and show that \textsc{Weakestmo2} satisfies \(LBRF_{RC11}\).

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

- \(po \cup rf\) is acyclic \hspace{1cm} (no-thin-air)
- \(hb; eco^2\) is irreflexive \hspace{1cm} (coherent)

**Theorem 3.12.** \textsc{Weakestmo2} satisfies \(LBRF_{RC11}\). \hspace{1cm} (See Appendix D for the full proof.)

**Proof Sketch.** We introduce a subclass of promise-free event structures for which \(cf \cap jo = \emptyset\) holds. It is easy to show that for \textsc{Weakestmo2}-consistent promise-free event structure \(S\) justified-from relation coincides with reads-from relation: \(S.jf = S.rf\). Therefore every extracted execution of a consistent promise-free event structure is \(RC11\)-consistent, since \(po \cup rf\) acyclicity follows immediately from \(po \cup jf\) acyclicity. Thus it suffices to show that every \textsc{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 \textsc{Weakestmo2}-consistent event structure \(S\) one can consider a sequence of events \(\{e_1, ..., e_n\}\) of \(S\) ordered according to some total order extending \((S.po \cup S.jf)^*\). 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 \(cf \cap jo\) 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 \textsc{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.
Weakestmo2 to with an example. Consider program lb-imm while issuing a write corresponds to our-of-order speculative execution of some store. An event can remains promise-free.

Composing LBRF_{RC11} with RC11’s DRF_{SC} theorem, we get DRF_{SC} for Weakestmo2.

**Corollary 3.13.** Weakestmo2 satisfies DRF_{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 \( S \triangleright G \).

(See Appendix 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 \( G + TC \xrightarrow{e} TC' \) where \( TC \) and \( TC' \) are traversal configurations and \( e \) is an event being traversed. Traversal configuration is a tuple \( \langle C, I \rangle \), where \( C \subseteq G.E \) is a set of covered events and \( I \subseteq G.W \) 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 \( \langle C, I \rangle \):

\[
\text{dom}(po ; [C]) \subseteq C \quad C \cap W \subseteq I \quad \text{dom}(rf ; [C]) \subseteq I \quad \text{dom}((rf \setminus po) ; ppo ; [I]) \subseteq I
\]

Giving the operational semantics of traversal \( G + TC \xrightarrow{e} TC' \), 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 \( \text{NO-BAIT-AND-SWITCH} \) 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 \( W(a, 1) \) in Thread 1. To simulate this action, we create the branch \( e_{11} \rightarrow e_{31} \rightarrow e_{21} \rightarrow e_{21} \rightarrow e_{21} \) (see Fig. 6) using the receptiveness property [Podkopaev et al. 2019, § 6.4]. Receptiveness allows us to pick arbitrary values for intermediate 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 \( e^2 \) to the event structure.

Subsequently, the traversal issues \( W(b, 1) \). At this point, the construction of Moiseenko et al. [2020] adds a branch \( e^1 \rightarrow e^3 \rightarrow e^5 \). 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 \( e^1 \rightarrow e^3 \rightarrow e^5 \). Doing so satisfies NO-BAIT-AND-SWITCH because the new branch differs from the previous branch only at the point of immediate conflict: \( e^1 \rightarrow e^3 \rightarrow e^5 \). Starting from this intermediate branch, it becomes possible to add the required branch \( e^3 \rightarrow e^1 \) which now has the event \( e^1 \) 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 it does not add any new behaviors. That is, for every Weakestmo-consistent execution \( G_{tgt} \) of \( P_{tgt} \), there exists some Weakestmo2-consistent execution \( G_{src} \) of \( P_{src} \) with the same behavior: \( B(G_{src}) = B(G_{tgt}) \).

In Appendix 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 \( a = (r_1 := x) \) and \( b = (y := r_2) \) is a sound source-to-source transformation.

**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 \( G_{tgt} \) of \( P_{tgt} \), we consider event structure \( S_{tgt} \), s.t. \( S_{tgt} \triangleright G_{tgt} \), and we built it.
incrementally from the initial event structure: \( S_{\text{init}}(P_{\text{tgt}}) \rightarrow S_{\text{tgt}} \). Following these steps, we will construct \( S_{\text{src}} \) and the required graph \( G_{\text{src}} \).

To simulate the target event structure construction step \( S_{\text{tgt}} \rightarrow S'_{\text{tgt}} \), 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 \( S_{\text{src}} \rightarrow S'_{\text{src}} \).

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 \( b^f : W(y, \beta) \) 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 \( R(x, \alpha) \) 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 \( b^f \) and \( w^f \) 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 \( a^i \) and \( b^i \). The construction justifies the \( a^i : R(x, y) \) event by choosing the \( S_{\text{src} \cdot \text{co}} \)-maximal non-conflicting write from \( S_{\text{src} \cdot \text{hf}} : S_{\text{src} \cdot \text{hb}} \) prefix of \( a^i \). Since instructions \( a \) and \( b \) are assumed to be independent, the choice of the value \( \gamma \) for the read \( a^i_1 \) cannot affect the value of the write \( b^i_1 : W(y, \beta) \).

The construction then proceeds by adding events to the target and source event structures until the target reaches the event \( a^f \). At this point, the construction adds another branch to the source consisting of events \( a^2_1 \) and \( b^2_2 \). Now, the event \( a^2_2 \) can have the required label \( R(x, \alpha) \) because there is already a justifying event \( w^i \) in the source event structure matching the target’s justification event \( w^f \). Finally, the write \( b^i_2 \) is announced to be equal to \( b^i_1 \).

What remains to be shown is that the two conflicting branches \( a^1_1 \rightarrow b^1_1 \) and \( a^2_2 \rightarrow b^2_2 \) satisfy the axiom NO-BAIT-AND-SWITCH. Indeed, they only differ at the point of the immediate conflict, and moreover, the read \( a^i_1 \) is justified from a \((S_{\text{src} \cdot \text{hf}} : S_{\text{src} \cdot \text{hb}})\)-preceding write. \( \square \)

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).

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.)
Alg 1. Main exploration algorithm

\begin{algorithm}
\begin{algorithmic}[1]
\Procedure{Visit}{\(P, G, \Pi\)}
\If \~\text{cons}_{RC11}^{porf}(G) \Then \Return \EndIf
\Switch{a \leftarrow \text{next}_P(\Pi)(G)}
\Case{a \leftarrow \bot}
\If{\Pi = \bot} \Output{"Exec OK"} \EndIf
\Case{a \in \emptyset} \Exit{"Erroneous execution"} \EndCase
\Case{a \in R} \EndCase
\For{w \in \text{GetRFs}(G, \Pi, a)}
\State \text{Visit}(P, \text{SetRF}(G, w, a), \Pi)
\EndFor
\EndSwitch
\Return W
\EndProcedure
\end{algorithmic}
\caption{Algorithm 1}
\end{algorithm}

Alg 2. Exploration of revisits

\begin{algorithm}
\begin{algorithmic}[1]
\Procedure{VisitRevisits}{\(P, G, \Pi, \Pi', a\)}
\If{\(\Pi' \neq \emptyset\)}
\State \(\langle Rs, \Pi_c \rangle \leftarrow (\emptyset, \emptyset) \Comment{In cert}\)
\Else \If{\(\Pi \neq \emptyset\)} \Comment{Cert OK}\State \(\langle Rs, \Pi_c \rangle \leftarrow \langle \text{CertRevs}(G, a), \{a\} \rangle\) \EndIf
\EndIf
\State \(\langle Rs, \Pi_c \rangle \leftarrow \langle \text{GetRevs}(G, a), \emptyset \rangle\)
\For{\(\langle w, r \rangle \in Rs\)}
\State \(G' \leftarrow \text{Restrict}(G, r, w) \cap \text{rng}[\{r\}; G.po] \Comment{Normal exec}\)
\State \(\Pi' \leftarrow \text{Promises}(G, r, w) \cup \Pi_c \cup \text{InCyc}(G, r))\)
\State \text{Visit}(P, G', \Pi')
\EndFor
\EndProcedure
\end{algorithmic}
\caption{Algorithm 2}
\end{algorithm}

\begin{itemize}
\item GenMC’s \texttt{Visit} procedure explores all consistent executions of a program \(P\) under RC11\(^3\) recursively. During the exploration, \texttt{Visit} maintains the current exploration graph \(G\) which is augmented with a revisit set \(G.T\) that records all reads in \(G\) whose reads-from edge can be changed. Initially, \texttt{Visit} is called with an empty execution graph.
\end{itemize}

At each step, as long as the current graph remains consistent (Line 2), \texttt{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 \texttt{error} (Line 7). Otherwise, it returns the event it added \(a\).

If \(a\) is a read, \texttt{Visit} has to consider all possible \(rf\) edges for it. To that end, for each possible \(rf\) option \(w\) (given by \texttt{GetRFs}), \texttt{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), \texttt{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 added to the graph. Thus, whenever \texttt{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 \texttt{VisitRevisits} (Line 14).

Otherwise, \texttt{Visit} simply recurses further.

\(^3\)Adapting the algorithm for a different model \(m\) merely requires changing the consistency check in Line 2 of Algorithm 1.
The \textit{VisitRevisits Procedure}. The calculation of revisitable reads is performed by \textit{VisitRevisits}, by means of \textit{GetRevs}. As can be seen in Algorithm 2, \textit{GetRevs} returns a set where \(a\) is paired with all revisitable reads to the same location that are not in its \textit{porf} prefix.

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

\textbf{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{align*}
    r_1 &:= x \quad & r_2 &:= y \\
    y &:= 1 \quad & x &:= 1
\end{align*}

(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 \textit{lb-nodep} highlights the difficulties arising when trying to enumerate the executions of programs with \textit{porf} cycles. In the presentation below, we omit the values read by reads in the graphs, as they can be deduced from the \textit{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{figure}[h]
\begin{center}
\begin{tikzpicture}
  \node at (0,0) [init] {\(0\)}; \node at (1,0) [init] {\(1\)}; \node at (2,0) [init] {\(2\)};
  \node at (0,-1) [init] {\(3\)}; \node at (1,-1) [init] {\(4\)}; \node at (2,-1) [init] {\(5\)};
  \node at (3,-1) [init] {\(6\)};
  \draw [->] (0) -- (1) node [midway, above] {\(rf\)};
  \draw [->] (1) -- (2) node [midway, above] {\(rf\)};
  \draw [->] (2) -- (3) node [midway, above] {\(rf\)};
  \draw [->] (2) -- (4) node [midway, above] {\(rf\)};
  \draw [->] (3) -- (5) node [midway, above] {\(rf\)};
  \draw [->] (4) -- (6) node [midway, above] {\(rf\)};
  \draw [->] (0) -- (1) node [midway, below] {\(R(x)\)};
  \draw [->] (1) -- (2) node [midway, below] {\(R(x)\)};
  \draw [->] (2) -- (3) node [midway, below] {\(R(x)\)};
  \draw [->] (2) -- (4) node [midway, below] {\(R(x)\)};
  \draw [->] (3) -- (5) node [midway, below] {\(R(x)\)};
  \draw [->] (4) -- (6) node [midway, below] {\(R(x)\)};
  \draw [->] (0) -- (5) node [midway, below] {\(W(y,1)\)};
  \draw [->] (1) -- (6) node [midway, below] {\(W(y,1)\)};
\end{tikzpicture}
\end{center}
\end{figure}

Note that, because there are no other reads-from options for \(R(x)\) except for 0, the loop in Line 9 in \textit{Visit} will only add one child node to the recursion tree. Similarly, because there are no revisit options for \(W(y, 1)\), \textit{VisitRevisits} is not executed, and GenMC only calls \textit{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, \textit{Visit} starts two recursively explorations: one where \(R(y)\) reads 0, and an alternative one where it reads from \(W(y, 1)\). Let us assume that \textit{Visit} first proceeds with the one where \(R(y)\) reads 0.

In a similar manner, when the write corresponding to \(x := 1\) is added to the graph, \textit{Visit} will initiate two recursive explorations: one where \(W(x, 1)\) does not revisit any reads, and one where it revisits \(R(x)\). Indeed, because \(R(x)\) is not \textit{porf}-before \(W(x, 1)\), it will be considered by \textit{GetRevs}, and \textit{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 5 below.

\begin{figure}[h]
\begin{center}
\begin{tikzpicture}
  \node at (0,0) [init] {\(0\)}; \node at (1,0) [init] {\(1\)}; \node at (2,0) [init] {\(2\)};
  \node at (3,0) [init] {\(3\)}; \node at (4,0) [init] {\(4\)}; \node at (5,0) [init] {\(5\)};
  \node at (6,0) [init] {\(6\)};
  \draw [->] (0) -- (1) node [midway, above] {\(rf\)};
  \draw [->] (1) -- (2) node [midway, above] {\(rf\)};
  \draw [->] (2) -- (3) node [midway, above] {\(rf\)};
  \draw [->] (2) -- (4) node [midway, above] {\(rf\)};
  \draw [->] (3) -- (5) node [midway, above] {\(rf\)};
  \draw [->] (4) -- (6) node [midway, above] {\(rf\)};
  \draw [->] (0) -- (1) node [midway, below] {\(W(x,1)\)};
  \draw [->] (1) -- (2) node [midway, below] {\(W(x,1)\)};
  \draw [->] (2) -- (3) node [midway, below] {\(W(x,1)\)};
  \draw [->] (2) -- (4) node [midway, below] {\(W(x,1)\)};
  \draw [->] (3) -- (5) node [midway, below] {\(W(x,1)\)};
  \draw [->] (4) -- (6) node [midway, below] {\(W(x,1)\)};
\end{tikzpicture}
\end{center}
\end{figure}
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 \textsc{Visit} does in the read case), $W(x, 1)$ would not have 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 \textsc{Visit} backtracks and explores the alternative \textit{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 \textit{rf} edge.

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

### 4.2 WMC: Model Checking under $\text{Weakestmo2}$

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, \textsc{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-prior 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 \textsc{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 $a$:

\[
\{r\} \triangleq ([R]; \equiv_{\text{loc}}; [W]) \cap (G.rpo; (G.rf \setminus G.po); G.porf) \setminus (\text{hb} \cup \text{hb}^{-1})
\]

Put differently, \textsc{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 visits a porf-earlier read $r$, the graph that \textsc{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.

\textsc{WMC} resolves this problem with a two-step approach. As a first step, when a write 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 \textsc{Promises} (Line 9), which is in turn used during the recursive exploration (Line 10). Promise sets, $\Pi$, 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 \textsc{Visit} encounters a non-empty promise set, WMC initiates a certification phase. During this phase, WMC operates in a restricted mode: all promises of $\Pi$ have

Proc. ACM Program. Lang., Vol. 6, No. OOPSLA2, Article 152. Publication date: October 2022.
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 $\Pi$ 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 $R(y)$ to be revisited by $W(y, 1)$, eventually leading to graph 2 below, where $W(x, 1)$ needs to be certified:

![Diagram]

Let us now see what happens when Visit proceeds with exploration corresponding to graph 2. Since $\Pi$ is non-empty, WMC enters a certification phase. The next event to be added is $W(x, 1)$, 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.

**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

---

\(^4\)The exploration is similar to the one presented for 2 above, and is thus omitted for brevity.
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.

\[
\begin{align*}
  r_1 &:= x \\
  \text{if } r_1 = 0 &\text{ then } \text{ if } r_2 = 0 &\text{ then (LB+CTRL) } \\
  y &:= 1 \\
  x &:= 1 \\
  r_2 &:= y \\
  r_1 &:= x \\
  r_2 &:= z \\
  y &:= 1 \\
  x &:= 1 \\
  z &:= 2 \\
\end{align*}
\]

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 \( \text{porf} \)-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 \( \Pi \).

Starting with the calculation of non-local options, when the certification is over, we have to a) calculate alternative \( \text{rf} \) 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 \( \text{po-after} \ r \) that are read externally, and are \( \text{porf} \)-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 sufficient. 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 \( \text{porf} \)-before \( w' \). Since \( w' \) is itself \( \text{porf} \)-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, \( \Pi_c \). (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:
\[
\begin{align*}
z &:= 1 \\
& \quad \begin{cases} r := z \\
s := x \lor s = 1 \text{ then } \\
y := 1 \end{cases} \\
a := y \\
x := a
\end{align*}
\]

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 \( \text{po} \)-before the cycle, and (2) ensure that the cycle will continue to exist after these revisits take place. Luckily, our definition of \text{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 \( \text{po} \)-before the cycle. To solve the second issue (i.e., guarantee that the cycle will exist after these revisits take place), we use the \text{InCyc} function, which returns all writes in the certification thread that are \( \text{po} \)-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.

- \text{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.

- \text{HMC} [Kokologiannakis et al. 2020] is an extension of \text{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.

- \text{HMC}_{\text{LBRF}} is a variant of HMC that we implemented in order to further demonstrate the benefits of LBRF. \text{HMC}_{\text{LBRF}} leverages LBRF and starts calculating dependencies only when it finds a load buffering race.

- \text{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).

- \text{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.

- \text{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

\(^5\)As a further optimization, \text{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 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 256 GB 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]. 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.

Table 1. Overhead of LBRF on GENMC benchmarks

<table>
<thead>
<tr>
<th></th>
<th>GENMC</th>
<th>WMC</th>
<th>HMC</th>
<th>HMC_LBRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB-racy</td>
<td>0.53</td>
<td>0.55</td>
<td>0.71</td>
<td>0.76</td>
</tr>
<tr>
<td>LB-race-free</td>
<td>111.82</td>
<td>134.09</td>
<td>235.85</td>
<td>155.06</td>
</tr>
</tbody>
</table>

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.

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.

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).

6We could not port all 43 of their benchmarks as they are written in C++, only a subset of which is supported by GENMC.
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_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_LBRF performs similarly to HMC. When LB races are not present though (e.g., for sigma), HMC_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_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 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.

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>GENMC</th>
<th>HMC</th>
<th>HMC_LBRF</th>
<th>NIDHUGG</th>
<th>RMEM</th>
<th>CDSCHECKER</th>
<th>WMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>reorder(2)</td>
<td>0.06</td>
<td>0.11</td>
<td>0.07</td>
<td>1.37</td>
<td>77.10</td>
<td>0.04</td>
<td>0.06</td>
</tr>
<tr>
<td>singleton</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.12</td>
<td>96.21</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>fib_bench</td>
<td>13.09</td>
<td>25.98</td>
<td>26.68</td>
<td>237.73</td>
<td>0.00</td>
<td>0.00</td>
<td>37.66</td>
</tr>
<tr>
<td>szymanski(1)</td>
<td>0.05</td>
<td>0.07</td>
<td>0.08</td>
<td>0.13</td>
<td>5.56</td>
<td>0.28</td>
<td>0.06</td>
</tr>
<tr>
<td>szymanski(2)</td>
<td>246.23</td>
<td>399.79</td>
<td>402.34</td>
<td>3.74</td>
<td>529.85</td>
<td>1688.60</td>
<td>321.23</td>
</tr>
<tr>
<td>szymanski(3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peterson(10)</td>
<td>0.03</td>
<td>0.06</td>
<td>0.07</td>
<td>0.65</td>
<td>0.25</td>
<td>0.14</td>
<td></td>
</tr>
<tr>
<td>peterson(20)</td>
<td>0.12</td>
<td>0.31</td>
<td>0.30</td>
<td>3.81</td>
<td>0.13</td>
<td>1.09</td>
<td></td>
</tr>
<tr>
<td>peterson(30)</td>
<td>0.32</td>
<td>0.84</td>
<td>0.85</td>
<td>13.54</td>
<td>3.86</td>
<td>4.43</td>
<td></td>
</tr>
<tr>
<td>dekker</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.14</td>
<td>30.09</td>
<td>0.23</td>
<td>0.02</td>
</tr>
<tr>
<td>sigma</td>
<td>141.56</td>
<td>352.86</td>
<td>168.94</td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>157.91</td>
</tr>
<tr>
<td>indexer(12)</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td></td>
<td>0.90</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>indexer(13)</td>
<td>0.05</td>
<td>0.07</td>
<td>0.07</td>
<td></td>
<td>116.71</td>
<td>0.05</td>
<td></td>
</tr>
<tr>
<td>indexer(14)</td>
<td>0.30</td>
<td>0.48</td>
<td>0.50</td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>0.34</td>
</tr>
<tr>
<td>indexer(15)</td>
<td>2.45</td>
<td>4.03</td>
<td>4.07</td>
<td></td>
<td>0.00</td>
<td>2.85</td>
<td></td>
</tr>
</tbody>
</table>
Table 3. Benchmarks adapted from Norris et al. [2013]

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>GenMC</th>
<th>HMC</th>
<th>HMC_{LBRF}</th>
<th>CDSChecker</th>
<th>WMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>barrier-bnd</td>
<td>1.29</td>
<td>2.67</td>
<td>1.60</td>
<td>4.15</td>
<td>1.34</td>
</tr>
<tr>
<td>mpmc-queue-bnd</td>
<td>1.87</td>
<td>4.18</td>
<td>2.14</td>
<td>19.78</td>
<td>2.09</td>
</tr>
<tr>
<td>treiber-stack-bnd</td>
<td>0.56</td>
<td>1.28</td>
<td>0.65</td>
<td>⊗</td>
<td>0.58</td>
</tr>
<tr>
<td>linuxrwlocks-bnd</td>
<td>0.29</td>
<td>0.57</td>
<td>0.38</td>
<td>⊗</td>
<td>0.46</td>
</tr>
<tr>
<td>seqlock-bnd</td>
<td>45.40</td>
<td>109.24</td>
<td>111.21</td>
<td>⊗</td>
<td>47.78</td>
</tr>
</tbody>
</table>

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 Mem 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.

### 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 \( \text{porf} \) 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 \( \text{rf} \) edge in the race’s \( \text{porf} \) 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 (zymanski) required 10 iterations. The reason is that zymanski 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-Pharbod 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 \( \text{LBF-CEX} \). We expect that their model satisfies LBRF and probably also CL, but have not yet investigated a proof of these conjectures.

<table>
<thead>
<tr>
<th></th>
<th># Execs</th>
<th>Dups</th>
<th>Blocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB+ctrl(10)</td>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LB+ctrl(12)</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LB+ctrl(14)</td>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LB+data(10)</td>
<td>1024</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LB+data(12)</td>
<td>4096</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LB+data(14)</td>
<td>16 384</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LB-nodep(10)</td>
<td>1024</td>
<td>0.9%</td>
<td>0.9%</td>
</tr>
<tr>
<td>LB-nodep(12)</td>
<td>4096</td>
<td>0.3%</td>
<td>0.3%</td>
</tr>
<tr>
<td>LB-nodep(14)</td>
<td>16 384</td>
<td>0.1%</td>
<td>0.1%</td>
</tr>
<tr>
<td>LB-pairs(10)</td>
<td>1024</td>
<td>205.2%</td>
<td>33.3%</td>
</tr>
<tr>
<td>LB-pairs(12)</td>
<td>4096</td>
<td>281.5%</td>
<td>33.3%</td>
</tr>
<tr>
<td>LB-pairs(14)</td>
<td>16 384</td>
<td>376.8%</td>
<td>33.3%</td>
</tr>
</tbody>
</table>
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


Model Checking on Multi-execution Memory Models


A PROOF OF THE LBRF THEOREM FOR IMM

In this section, we prove that IMM [Podkopaev et al. 2019] provides the LBRF_{RC11} guarantee. IMM is an intermediate memory consistency model that abstracts over implementation details of hardware memory models, in particular models of x86, ARMv7, ARMv8, and POWER.

The crucial property of IMM required for our proof is the fact that the model tracks syntactic dependencies between the events and defines a subset of the program order that is preserved (ppo). All typical hardware memory models also satisfy this property, which suggests that they can also be shown to satisfy LBRF with a similar argument.

Before diving deep into the details of the proof, we provide a quick recap of the IMM model. We refer readers to Podkopaev et al. [2019] for a more detailed presentation of the model.

Definition A.1. An IMM execution graph $G$ is an execution graph augmented with data, control, address and CAS dependency relations $\langle \text{data}, \text{ctrl}, \text{addr}, \text{casdep} \rangle$. The dependency relation is the union of the above:

$\text{deps} \triangleq \text{data} \cup \text{ctrl} \cup \text{addr} \cup \text{casdep}$

Definition A.2. The relation barrier-ordered-before is defined as follows:

$\text{bob} \triangleq \text{po} \cup \text{Wrel} \cup \text{Racq} \cup \text{po} \cup \text{Wrel} \cup \text{po} \cap \text{loc} \cup \text{Wrel}$

Definition A.3. Preserved program order is defined as follows:

$\text{ppo} \triangleq \text{R} \cup (\text{deps} \cup \text{rfi})^+ \cup \text{Wrel}$

Definition A.4. Detour relation is defined as follows:

$\text{detour} \triangleq (\text{coe} \cup \text{rfe}) \cap \text{po}$

Definition A.5. The global acyclic ordering is defined as follows:

$\text{ar} \triangleq \text{rfe} \cup \text{bob} \cup \text{ppo} \cup \text{detour} \cup \text{pscF}$

Similarly to Weakestmo and RC11, the IMM model also defines the relations $\text{sw}$, $\text{hb}$, $\text{scb}$, $\text{pscF}$, $\text{pscF}_{\text{base}}$, and $\text{psc}$. However, the IMM has a slightly different definition of $\text{sw}$, which also affects other relations depending on it. For the purpose of our proof this discrepancy does not matter. What does matter is that the IMM definitions are stronger than RC11 ones, that is $\text{sw}_{\text{RC11}} \subseteq \text{sw}_{\text{IMM}}$, $\text{hb}_{\text{RC11}} \subseteq \text{hb}_{\text{IMM}}$. In the rest of this section, we omit the subscripts whenever the version of the definition used (i.e., IMM or RC11 one) is clear from the context. The exact definitions of IMM versions of these relations and the motivation behind them can be found in [Podkopaev et al. 2019].

Definition A.6. Execution graph $G$ is IMM-consistent if the following hold:

- $\text{ar}$ is acyclic
- $\text{hb}_{\text{IMM}} \cup \text{eco}$ is irreflexive
- $\text{rmw} \cap (\text{fr} \cup \text{co}) = \emptyset$
- $\text{psc}$ is acyclic

Definition A.7 (porf-sequence). Given an execution graph $G$, a set of events $E \subseteq G. E$ is a porf-sequence if there exists a linearization $\{e_1, ..., e_n\} = E$, s.t. for all $1 \leq i < n$ either events $e_i$ and $e_{i+1}$ are either immediate po neighbors or $e_{i+1}$ reads externally from $e_i$:

$\langle e_i, e_{i+1} \rangle \in \text{po}_{\text{imm}} \cup \text{rfe}$

Definition A.8 (porf-cycle). A set of events $E \subseteq G. E$ is a porf-cycle if it is a closed porf-sequence, i.e., $\langle e_n, e_i \rangle \in \text{po}_{\text{imm}} \cup \text{rfe}$. 

Proc. ACM Program. Lang., Vol. 6, No. OOPSLA2, Article 152. Publication date: October 2022.
Definition A.9 ( porr -span). Let \( E \) be a porr -cycle of graph \( G \). Its span is the subsequence \( S = E \cap (\text{dom}(G.\text{rfe}) ; [E]) \cup \text{rng}([E] ; G.\text{rfe})) \). In other words, the span contains writes read externally by some event in the cycle, and read events reading externally from the cycle.

Lemma A.10. \( G \) be an IMM-consistent graph of program \( P \) and let \( E \) be a porr -cycle, s.t. the porr -prefix of the cycle does not contain other cycles, i.e., for \( X = \text{dom}(G.\text{porf}) ; [E]) \setminus E \) the graph \( G|_{\text{ex}} \) is porr -acyclic. Then, there exists a RC11-consistent execution graph \( G' \) of \( P \) with an LB race.

Proof. First, note that the cycle \( E \) cannot consist only of \( \text{po} \) or only of \( \text{rfe} \) edges. In the former case, it would contradict the fact that \( \text{po} \) is partial order and thus it is acyclic. In the latter case, we have that \( \text{rfe} ; \text{rfe} = \emptyset \), since for \( \text{rfe} \) its domain (write events) and codomain (read events) are disjoint sets.

Second, the cycle \( E \) can contain several smaller subcycles. Below we will show that these subcycles can be eliminated incrementally one-by-one yielding a sequence of IMM consistent graphs each one having smaller number of cycles, and that upon the disposal of the last cycle we will arrive to porr acyclic RC11 consistent graph with LB race.

Consider the span \( S \) of \( E \). We will show that it is always possible to find an edge \( \langle r, w \rangle \) and event \( w' \) in the span \( S \), s.t. the following is true:

\[
\begin{align*}
& r \in G.R^{1x} \setminus G.R_{\text{ex}} \text{ and } w \in G.W^{1x} \\
& \langle r, w \rangle \in G.\text{po} \setminus (G.\text{ppo} \cup G.\text{bob}) \\
& \text{dom}([G.R_{\text{loc}(r)} \cap S] ; G.\text{po} ; [r]) \text{ is empty} \\
& \langle w', r \rangle \in G.\text{rfe} \\
& \langle w', r \rangle \notin \text{hb}_{\text{IMM}} \cup G.\text{hb}^{-1}_{\text{IMM}}
\end{align*}
\]

We do this by induction on the length \( n \) of the span \( S \).

Base Case: \( P(i = 0) \).
In this case we immediately arrive at contradiction. The fact that the span of the cycle is empty implies that the cycle consists only of \( \text{po} \) edges, which cannot be true as we have mentioned above.

Induction Step: \( P(i) \Rightarrow P(i + 1) \).

First, note that it cannot be the case that the span consists only of \( \text{ppo} \), \( \text{bob} \) and \( \text{rfe} \) edges, that is the following is false:

\[
\begin{align*}
& \langle e_i, e_{i+1} \rangle \in \text{ppo} \cup \text{bob} \cup \text{rfe} \text{ for all } 1 \leq i < n; \\
& \langle e_n, e_1 \rangle \in \text{ppo} \cup \text{bob} \cup \text{rfe}.
\end{align*}
\]

Otherwise we would get the \( \text{ppo} \cup \text{bob} \cup \text{rfe} \) cycle, which is forbidden by IMM consistency.

Therefore, there should exists at least one edge \( \langle a, b \rangle \in \text{po} \setminus (\text{ppo} \cup \text{bob}) \). Then consider the cases for \( a \) and \( b \).

First, assume \( a \in G.R \) and \( b \in G.W \). Then note that it should be the case that \( a \in G.R^{1x} \setminus G.R_{\text{ex}} \) since otherwise we would have \( \langle a, b \rangle \in [G.R^{\text{acq}}] \setminus \text{po} \subseteq \text{bob} \). Similarly if \( a \in G.R_{\text{ex}} \) then \( \langle a, b \rangle \in [G.R_{\text{ex}}] ; \text{po} \subseteq \text{ppo} \). By the same argument \( w \in G.W^{1x} \), because otherwise \( \langle a, b \rangle \in \text{po} ; [W^{\text{rel}}] \subseteq \text{bob} \).

Second, suppose there exists \( c \in G.R_{\text{loc}(a)} \cap S \), s.t. \( \langle c, a \rangle \in G.\text{po} \). Then we can construct a span of smaller length with an edge \( \langle c, b \rangle \), by inductive assumption, should contain edge \( \langle r, w \rangle \) satisfying our constraints. Therefore, there exists no such \( c \).

Then let \( w_a \) be a write event of the span \( S \), s.t. \( \langle w_a, a \rangle \in G.\text{rfe} \). Suppose \( \langle w_a, a \rangle \in \text{hb}_{\text{IMM}} \cup \text{hb}^{-1}_{\text{IMM}} \). If \( \langle w_a, a \rangle \in \text{hb}_{\text{IMM}} \) then consider the last edge in \( \text{hb}_{\text{IMM}} \) path. It cannot be \( \text{sw}_{\text{IMM}} \) edge since it would imply \( a \in G.R^{\text{acq}} \) which cannot be true as we have shown above. If the last event is \( \text{po} \) edge, then there should exists \( c \in \text{rng}([w_a] ; G.\text{rfe}) \) and \( \langle c, a \rangle \in G.\text{po} \). But we have proven there exists no such \( c \). Next assume \( \langle a, w_a \rangle \in \text{hb}_{\text{IMM}} \). Then there should exists \( d \in W \), s.t. \( \langle a, d \rangle \in \text{po} \), and either \( d \) is a release write or there exists release fence between \( a \) and \( d \). Then there should exists...
write $d'$ belonging to a span $S$, s.t. $(d, d') \in (\text{ppo} \cap =_{\text{1oc}})^3$. Then again we can construct a smaller span with an edge $(a, d')$ and by induction get suitable edge $(r, w)$. If neither $(w_a, a) \in \text{hb}_{\text{IMM}}$ nor $(a, w_a) \in \text{hb}_{\text{IMM}}$, we have $r = a$, $w = b$ and $w' \in \text{dom}(\text{G.rf} ; [a])$ which satisfy our constraints.

Otherwise, let, for example $a \in \text{G.R}$ and $b \in \text{G.R}$. Let $w_a$ is write from which a reads, and $w_b$ is a write from which $b$ reads. By definition of the span, both writes are external writes belonging to the porf cycle. But then for the cycle $S$ to be closed, there should exist $w', r' \in S$, s.t. $(w', r') \in \text{G.rfe}$, $(b, w') \in \text{ppo}$, and $(r', w_b) \in \text{G.porf}$. Then we would be able to construct a smaller porf span from $w'$ going through $r'$ to $b$ without visiting $a$. By induction this span should contain the required edge $(r, w)$. Other cases for $a$ and $b$ can be handled similarly.

As a result, we get an edge $(r, w)$ and write $w'$ satisfying the constraints given above. Note that by definition of bob we can derive that $(r, w) \in \text{rpo}$.

Next, using the receptiveness property one can construct an IMM consistent graph $\overline{G}$, s.t. $G \equiv \overline{G}$ in the following sense

- $G.\text{lab}|_X = \overline{G}.\text{lab}|_X$ where $X \triangleq G.\text{E} \setminus \text{rng}([r] ; G.\text{ppo}^2 ; G.\text{rf}^1)$
- $\overline{G}.\text{E} = G.\text{E}$
- $\overline{G}.\text{po} = G.\text{po}$
- $\overline{G}.\text{rf}|_{G.\text{E}\setminus\{r\}} = G.\text{rf}|_{G.\text{E}\setminus\{r\}}$
- $\overline{G}.\text{rfe} ; [r] \subseteq G.\text{rfe}^2 ; G.\text{hb}_{\text{IMM}}$
- $\overline{G}.\text{co} = G.\text{co}$

Put simply, it is possible to construct a graph similar to the original one, that would only be different in the choice of reads-from write for the read $r$, and the labels of events that depend on $r$ via preserved program order. The formal definition of receptiveness property and the details on graph construction can be found in [Podkopaev et al. 2019, §6.4]. Below we give some intuitive explanation. Since the ppo relation includes the dependencies relations and reads-from relation, which together capture the control and data flow in intra-thread computation, the value of the read (and its reads-from source) can be safely replaced with a different value, and this change will affect only the ppo descendants of the read. Since we have that $(r, w) \notin G.\text{ppo}$, the label of $w$ would not be affected if we’ll redirect the read $r$ to read-from $G.\text{rfe}^2 ; G.\text{hb}_{\text{IMM}}$ prior write. Redirecting to such write is safe, since it cannot violate axiom IMM-COHERENT. Also, because $r \notin G.\text{R}_\text{ex}$, a redirection of this read cannot violate RMW-ATOMIC.

Therefore we have an IMM consistent graph $\overline{G}$, which has one porf cycle less.

It left to note that upon the elimination of the last cycle, we would have edge $(r, w)$ and a write event $w'$, s.t.:

- $\text{loc}(r) = \text{loc}(w')$
- $(r, w) \in \overline{G}.\text{rpo}$
- $(w, w') \in \overline{G}.\text{rfe} ; \overline{G}.\text{porf}$
- $(w', r) \in \overline{G}.\text{hb}_{\text{IMM}} \cup \overline{G}.\text{hb}_{\text{IMM}}^{-1}$

That is, the pair $(r, w')$ forms an LB race. Since the sequence $E$ does not form a cycle in $\overline{G}$, and its porf prefix $P \triangleq \text{dom}(\overline{G}.\text{porf} ; \{E\})$ also contains no cycles by our assumption, we have that $\overline{G}|_P$ is porf acyclic. Since it is also IMM consistent, we have that it also satisfies axioms COHERENT, RMW-ATOMIC, IMM-SEQUENTIAL-CONSISTENCY (in particular, it is implied by the fact that IMM has a stronger notion of happens-before $\text{hb}_{\text{RC11}} \subseteq \text{hb}_{\text{IMM}}$). Therefore $\overline{G}$ is a RC11-consistent graph of $P$ and $(r, w')$ is an LB race.

\textbf{THEOREM A.11.} Let $P$ be LB-race-free under RC11. Then the set of its RC11 consistent execution graphs coincides with the set of its IMM consistent graphs.
Proof. The most challenging part of the proof has already been done in A.10. Let \( G \) be some IMM consistent graph of \( P \) and assume it is not RC11 consistent, i.e., it contains some \( G.\text{porf} \) cycles. Then we can order non-overlapping \( G.\text{porf} \) cycles: the cycle \( E_1 \) is said to be less than \( E_2 \) if \( E_1 \subseteq \text{dom}(G.\text{porf} ; [E_2]) \). It is easy to show that this definition gives us partial order. Next pick an arbitrary minimal cycle \( E \). Because its minimal w.r.t. to the partial order defined above, its \( G.\text{porf} \) prefix cannot contain another \( G.\text{porf} \) cycle. Therefore the preconditions of lemma A.10 are met and we can construct a RC11 consistent execution graph with LB race, which contradicts our assumption that \( P \) is LB-race-free. □

B DEFINITION OF Promising WITH CERTIFICATION LOCALITY

In this section, we define a version of the Promising semantics that satisfies certification locality. For conciseness, we present only the relaxed fragment of the Promising machine, since its extra machinery to handle release/acquire synchronization, RMW operations, and fences is orthogonal to our adaptations.

Promising represents memory, \( M \), as a set of messages, where each message \( m = \langle x : v@t \rangle \) is a tuple containing a location \( x \), a value \( v \), and a timestamp \( t \in \mathbb{Q} \) and represents a write event. Besides the memory, the state of the operational semantics also contains \( TS \), which is a map from thread identifiers to the thread-local state of each thread. This thread-local state, \( TS \), contains three components: (1) the actual thread-local state, \( \sigma \), containing the values of the program counter and the local registers, which is modeled abstractly; (2) the thread view, \( V \), a map from locations to timestamps, recording the latest write for each location that the thread is aware of, i.e., has read or executed itself; and (3) a set, \( P \), of messages that the thread has promised.

In our adaptation, we change this last component of the thread state, \( P \), to be a partial function from messages to sets of messages. The purpose of this change is to represent the set of external writes that are performed in the certification of the promise so that the actual execution performs the exact same set of external writes before fulfilling the promise.

The top-level rule of Promising remains unchanged:

\[
\begin{align*}
TS = TS(i) & \\
\langle TS, M \rangle \rightarrow \langle TS', M' \rangle & \\
\langle TS', M' \rangle \overset{\text{promise-free}}{\rightarrow} \ast & \\
\langle TS', M' \rangle & \rightarrow \langle TS''', M''' \rangle
\end{align*}
\]

(Machine Step)

The machine can make a step, if some thread \( i \) makes a step to some configuration \( \langle TS', M' \rangle \) for which all outstanding promises are certifiable. That is, there exists a promise-free execution of that thread that reaches a configuration \( \langle TS''', M''' \rangle \) with no outstanding promises.

We move on to the thread-local transitions. The first rule concerns ‘silent’ transitions that do not interact with memory. Here and in subsequent rules, Promising assumes a labeled transition relation saying how to update the values of the registers, \( \sigma \overset{\ell}{\rightarrow} \sigma' \).

\[
\sigma \overset{\ell}{\rightarrow} \sigma' \rightarrow \langle \langle \sigma, V, P \rangle, M \rangle \rightarrow \langle \langle \sigma', V, P \rangle, M \rangle
\]

(Silent)

The next two rules concern reads. While the original Promising semantics has a single rule for handling this case, we split it into two rules to distinguish between local and external reads.

\[
\sigma \overset{R(x,a)}{\rightarrow} \sigma' \rightarrow V(x) = t \rightarrow \langle \langle \sigma', V, P \rangle, M \rangle
\]

(Read-Local)

The rule for external reads checks that all outstanding promises of the thread allow reading the not previously observed message \( m \), and removes \( m \) from the sets of external messages recorded in the \( P \) component.
\[
\sigma \xrightarrow{R(x,0)} \sigma' \quad V(x) < t \quad \forall p \in \text{dom}(P), \ m \in P(p) \\
m = \langle x : v \oplus t \rangle \in M \quad P' = \{\langle p, W \setminus \{m\} \rangle | \langle p, W \rangle \in P\} \\
\langle\sigma, V, P, M\rangle \rightarrow \langle\sigma', V[x \mapsto t], P', M\rangle \quad \text{(Read-External)}
\]

Next is the rule for promising a message. Promising is essentially unconditional: all that is required is for the message to not already exist in memory. In practice, of course, promises are constrained by the top-level rule, which requires all promises to be certifiable, and so only promises guided by the subsequent instructions of the thread have a chance of being certified.

\[
M' = M \cup \{m\} \quad P' = P \cup \{\langle m, W \rangle\} \\
\langle\sigma, V, P, M\rangle \rightarrow \langle\sigma', V[x \mapsto t], P', M'\rangle \quad \text{(Promise)}
\]

The next rule is for a write that fulfills an outstanding promise. This rule is also used for normal (non-promised) writes: we simply precede it with the step that promises the necessary write.

\[
\sigma \xrightarrow{W(x,0)} \sigma' \quad V(x) < t \quad \langle m, \emptyset \rangle \in P \\
m = \langle x : v \oplus t \rangle \quad P' = P \setminus \{\langle m, \emptyset \rangle\} \\
\langle\sigma, V, P, M\rangle \rightarrow \langle\sigma', V[x \mapsto t], P', M\rangle \quad \text{(Fulfill)}
\]

Note that this rule requires the fulfilled promise to have associated an empty set of external messages, that is, all associated external messages need to be already have read.

The final rule allows us to extend the set of external reads attached to outstanding promises by a single additional message.

\[
m_1, \ldots, m_k \in \text{dom}(P) \quad P' = P[m_1 \mapsto P(m_1) \cup \{m'\}] \ldots [m_k \mapsto P(m_k) \cup \{m'\}] \\
\langle\sigma, V, P, M\rangle \rightarrow \langle\sigma', V, P', M\rangle \quad \text{(Extend-Promise)}
\]

Promise-free thread transitions are all the above, except for the (Extend-Promise) step. It can be shown that if there is a sequence of thread-local steps that reaches a configuration without any promises, that sequence can be rearranged so that every (Promise) step is immediately before the corresponding (Fulfill) step.

C COMPLETE DEFINITION OF WEAKESTMO2

Definition C.1. An event is a tuple \langle id, tid, lab \rangle where id \in \mathbb{N} is a unique identifier for the event, tid \in \mathbb{N} identifies the thread to which the event belongs, and lab is a label of the form:

- R^o(x,v) for a read of v \in Val from x \in Loc with mode o \in \{ex, non-ex\} \times Mod;
- W^o(x,v) for a write of v \in Val to x \in Loc with o \in \{ex, non-ex\} \times Mod;
- F^o for a fence with mode o \in Mod,

where Mod = \{rlx, acq, rel, acqrel, sc\}. The modes are partially ordered by \sqsubseteq as follows:

\[
\begin{align*}
\rlx & \sqsubseteq \rel \\
\rlx & \sqsubseteq \acq \\
\rlx & \sqsubseteq \acqrel \sqsubseteq \sc
\end{align*}
\]

Definition C.2. An event structure is S is a tuple \langle E, po, jf, ew, co \rangle where:

- E \subseteq \text{Event} is a set of events.
- po \subseteq E \times E is program order relation.
- jf \subseteq E \times E is justified from relation.
- ew \subseteq E \times E is equal-writes relation.
- co \subseteq E \times E is coherence order relation.

Definition C.3. Conflict relation is defined as follows: cf \triangleq =_{\text{tid}} \setminus (po \cup po^{-1})^2
**Definition C.4.** Immediate program order and immediate conflict relations are defined as follows:

\[
p_{\text{imm}} \triangleq p_{\text{po}} \setminus (p_{\text{po}} \cdot p_{\text{po}}) \\
\text{cf}_{\text{imm}} \triangleq \text{cf} \setminus (\text{cf} \cdot p_{\text{po}} \cup p_{\text{po}}^{-1} \cdot \text{cf})
\]

**Definition C.5.** Weakest event structure is well-formed if the following conditions are met:

1. There exists the initialization event \( e_0 \in E \).
   
   We assume that: \( e_0 = (0, \bot, W_{\text{rlx}}(\bot, 0)) \) for \( f \in \{\text{tid}, \text{lab}, \text{loc}\} \) we define:
   
   \[
   <_f \triangleq \{e_0\} \times (E \setminus \{e_0\}) \\
   \leq_f \triangleq \{e_0\} \times (E \setminus \{e_0\}) \cup =_f
   \]

2. \( p_{\text{po}} \) is strict partial order.
3. \( p_{\text{po}} \) orders events from the same thread (the initialization event is placed before other events):
   
   \[
   <_{\text{tid}} \subseteq p_{\text{po}} \subseteq \leq_{\text{tid}}
   \]
4. \( p_{\text{po}} \) is downward-total: \( p_{\text{po}} \cdot p_{\text{po}}^{-1} \subseteq (p_{\text{po}} \cup p_{\text{po}}^{-1}) \).
5. Immediate conflict can only appear between relaxed reads: \( \text{cf}_{\text{imm}} \subseteq \text{R}_{\text{ex}} \times \text{R}_{\text{ex}} \)
6. Every write-exclusive has a corresponding read part:
   
   \( W_{\text{ex}} \subseteq \text{rng}([\text{R}_{\text{ex}}]; (p_{\text{po}} \cdot \text{imm} \cap =_{\text{loc}})) \)
7. \( jf \) relates write/read pairs of accesses to same location with the same value.
   
   \[
   jf \subseteq [W]; (\leq_{\text{loc}} \cap =_{\text{val}}); [R]
   \]
8. \( jf^{-1} \) is functional: \( jf; jf^{-1} \subseteq [W] \)
9. Every read event has a justification event: \( R \subseteq \text{rng}(jf) \)
10. \( ew \) is a symmetric relation.
11. \( ew \) relates conflicting write events with the same location and value.
    
    \[
    ew \subseteq [W_{\text{rlx}}]; (\text{cf} \cap =_{\text{loc}} \cap =_{\text{val}}); [W_{\text{rlx}}]
    \]
12. \( co \) is strict partial order.
13. \( co \) orders write events with the same location (the initialization event is placed before other events).
    
    \[
    [W]; <_{\text{loc}} \subseteq [W] \subseteq co \subseteq [W]; \leq_{\text{loc}} \subseteq [W]
    \]
14. \( co \) does not link the writes from the same \( ew^* \) class:
    
    \[
    co \cap ew^* \subseteq \emptyset
    \]
15. \( co \) is closed w.r.t. \( ew^* \) equivalence classes:
    
    \[
    ew^*; co; ew^* \subseteq co
    \]
16. \( co \) is total on writes to the same location w.r.t. \( ew^* \) as equivalence relation:
    
    \[
    \forall x \in \text{Loc}. \ W_{x} \times W_{x} \subseteq ew^* \cup co \cup co^{-1}
    \]

Given an event structure, we define the following auxiliary relations:

\[
rf \triangleq (ew^*; jf) \setminus \text{cf} \quad \text{(reads-from)}
\]
\[
jfi \triangleq jf \cap po \quad \text{and} \quad rfi \triangleq rf \cap po \quad \text{(internal jf/rf)}
\]
\[
jfe \triangleq jf \setminus po \quad \text{and} \quad rfe \triangleq rf \setminus po \quad \text{(external jf/rf)}
\]
\[
jo \triangleq jfe; (po \cup jf)^* \quad \text{(justification order)}
\]
\[
\text{lbpat} \triangleq cf_{\text{imm}}; [\text{rng}(jf \cap (jf^*; \text{hb}))]; po \quad \text{(load buffering pattern)}
\]
For event $e \in S.E$, we call the set \( \text{dom}(jo; [e]) \) the justification set of $e$.

Definition C.6. A justified configuration $C$ of an event structure $S$ is a conflict-free, $rf$-complete, po-prefix-closed subset of its events. That is, $C \subseteq S.E$ and $[C] ; cf ; [C] = \emptyset$ and $dom(po ; [C]) \subseteq C$ and $C \cap R \subseteq rng([C] ; rf)$.

Definition C.7. An execution graph $G$ is extracted from an event structure $S$, denoted as $S \triangleright G$, if there exists a justified configuration $C$ s.t. $G = S|_{C}$ (in particular, $G.rf = S.rf|_{C}$).

Definition C.8. Event structure $S$ is Weakestmo2-consistent if the following conditions are met.

- $ecf$ is irreflexive. (NON-CONTRADICTORY)
- $jf \cap ecf = \emptyset$ (WELL-JUSTIFIED)
- $po \cup jf$ is acyclic (NO-THIN-AIR)
- $hb ; eco$ is irreflexive. (COHERENT)
- $(jfe ; [E^{acq}] \cup [F]) ; po ; ew^+ \subseteq (jfe ; [E^{acq}] \cup [F]) ; po$ (WELL-FENCED)
- $(cf \cap (jo \cup jo^{-1}))^+$ (CERTIFIED)
- $ew \subseteq (cf \cap (jo \cup jo^{-1}))^+$ (GROUNDED)
- $(jf \setminus (jf^+ ; hb)) ; po ; ew \subseteq jf ; (po \cup lbpat)$ (NO-BAIT-AND-SWITCH)

Definition C.9. Execution graph $G$ is Weakestmo2-consistent if there exists a Weakestmo2-consistent event structure s.t. $S \triangleright G$ and additionally $G$ satisfies the following constraints:

- $G.rmw \cap (G.fr ; G.co) = \emptyset.$ (RMW-ATOMIC)
- $G.psc$ is acyclic. (SEQUENTIAL-CONSISTENCY)

D PROOF OF THE LBRF THEOREM FOR A CORE FRAGMENT OF WEAKESTMO2

In this section, we prove that the fragment of Weakestmo2 without read-modify-writes and SC features (i.e., with only relaxed and release/acquire reads, writes and fences) provides the LBRF\textsubscript{RC11} guarantee. This corresponds to the version of the model presented in §3 extended with release/acquire accesses.

We postpone the proof of the theorem for the full model to Appendix E because the other features of the model are mostly orthogonal to the key ideas of the proof and yet they bring several technical difficulties, which require a refinement of LB race definition or a strengthening of the model.

Definition D.1. Execution graph $G$ is RC11-consistent if the following hold:

- $po \cup rf$ is acyclic (RF-NO-THIN-AIR)
- $hb ; eco$ is irreflexive. (COHERENT)
**Lemma D.2.** Every RC11-consistent execution graph is Weakestmo2-consistent.

**Proof.** Let $G$ be a RC11-consistent graph. Let $S = G$ (i.e., with $S.jf = G.rf$ and $G.cf = \emptyset$). Clearly, $S \supset G$. Therefore, we only need to show that $S$ is a Weakestmo2-consistent event structure.

Since $S.cf = \emptyset$, axioms NON-CONTRADICTION, WELL-JUSTIFIED are trivially satisfied. By well-formedness, we also have that $S.\text{ew} = \emptyset$ and therefore WELL-FENCED, CERTIFIED, GROUNDED, and NO-BAIT-AND-SWITCH are satisfied as well.

Since $G$ satisfies COHERENT, then $S$ also satisfies this constraint.

Finally, $(S.po \cup S.jf) = (G.po \cup G.rf)$ and thus its acyclicity follows directly from RF-NO-THIN-AIR. \qed

**Definition D.3.** A po-edge between a read and a write in an event structure is reorderable, if it connects two relaxed accesses and there is no release or acquire fence in between.

$$rpo \triangleq [R^{\text{rx}}_1] ; (po \setminus (po ; [F] ; po)) ; [W^{\text{rx}}_1]$$

**Definition D.4.** Given a well-formed execution graph $G$, a pair of events $(e_1, e_2)$ is called a load buffering race if it satisfies the following:
- $G.\text{loc}(e_1) = G.\text{loc}(e_2)$
- $(e_1, e_2) \notin G.\text{hb} \cup G.\text{hb}^{-1}$
- $(e_1, e_2) \in [R^{\text{rx}}_1] ; rpo ; rf \text{e} ; (po \cup rf)^* ; [W]$

**Definition D.5.** A program $P$ is said to be LB-race-free under RC11 if no RC11 consistent execution graph of $P$ contains a load buffering race.

**Definition D.6.** Event structure $S$ is prefix-closed if $\forall e \in S.\text{e} \text{ have } [e]_{(po \cup jf)^*} \subseteq S.\text{e}$ where $[e]_r \triangleq \text{dom}(r ; [e])$

**Definition D.7.** Given event structure $S$ and subset of its events $E \subseteq S.\text{e}$ we call $S|_E \triangleq (E, S.po|_E, S.jf|_E, S.\text{ew}|_E, S.co|_E)$ the restriction of $S$ onto $E$.

**Lemma D.8.** For any well-formed $S$ and any $E \subseteq S.\text{e}$ if $S|_E$ is prefix-closed then $S|_E$ is well-formed as well.

**Proof.** Follows directly by observing that all well-formedness properties are preserved under the restriction to prefix-closed subset of events. \qed

**Definition D.9.** Event structure $S$ is promise-free-consistent if it satisfies all the consistency constraints (see Def. C.8) except WELL-FENCED, CERTIFIED, GROUNDED, and NO-BAIT-AND-SWITCH.

**Lemma D.10.** For any promise-free-consistent $S$ and any $E \subseteq S.\text{e}$ if $S|_E$ is prefix-closed then $S|_E$ is promise-free-consistent as well.

**Proof.** Follows directly from the fact that properties NON-CONTRADICTION, WELL-JUSTIFIED, NO-THIN-AIR, and COHERENT are monotone, in the sense that if they hold for $S$ then they hold for any restriction of $S$. \qed

**Definition D.11.** Event structure $S$ is promise-free if it satisfies the following constraint.
- $\text{cf} \cap \text{jo} = \emptyset$ (promise-free)

**Lemma D.12.** If $S$ is PROMISE-FREE then $\text{cf} \cap (po \cup jf)^* = \emptyset$

**Proof.** First note that for two arbitrary relations $r_1, r_2$ the following equation holds:

$$(r_1 \cup r_2)^* = r_1^* \cup r_2^* ; (r_2 ; r_1^*)$$
By substitution \( r_1 \mapsto \text{po} \) and \( r_2 \mapsto \text{jf} \setminus \text{po} \) we get (note that since \( \text{po} \) is partial order we have \( \text{po}^+ = \text{po} \) and \( \text{po}^* = \text{po}^2 \)):

\[
\text{cf} \cap (\text{po} \cup \text{jf})^+ = \text{cf} \cap (\text{po} \cup \text{jf} \setminus \text{po})^+ = \text{cf} \cap (\text{po} \cup \text{po}^2 ; ((\text{jf} \setminus \text{po}) ; \text{po}^2)^*)
\]

\( \text{cf} \cap \text{po} = \emptyset \) by the definition of \( \text{cf} \). Observing that \( (\text{jf} \setminus \text{po}) ; \text{po}^2 \subseteq \text{jo} \) and that \( \text{jo} \) is transitive we left to show that \( \text{cf} \cap (\text{po}^2 ; \text{jo}) = \emptyset \). Suppose we have three events \( e_1, e_2, \) and \( e_3 \) s.t. \( (e_1, e_3) \in \text{cf}, (e_1, e_2) \in \text{po}^2 \) and \( (e_2, e_3) \in \text{jo} \). But since conflict propagates along program order, i.e., \( \text{cf} ; \text{po} \subseteq \text{cf}, \) we also have \( (e_2, e_3) \in \text{cf} \), which contradicts our assumption that \( \text{cf} \cap \text{jo} = \emptyset \).

**Lemma D.13.** If \( S \) is consistent and promise-free then \( S.\text{ew} = \emptyset \)

**Proof.** By *grounded* we have \( \text{ew} \subseteq (\text{cf} \cap (\text{jo} \cup \text{jo}^{-1}))^+ \). By *promise-free* we also have \( \text{cf} \cap (\text{jo} \cup \text{jo}^{-1}) = \emptyset \) which implies \( \text{ew} \subseteq \emptyset \).

**Lemma D.14.** If \( S \) is consistent and promise-free then \( S.\text{jf} = S.\text{rf} \)

**Proof.** Follows trivially from the definition of \( \text{rf} \), lemma D.13 and well-justified axiom.

\( \text{rf} \triangleq (\text{ew}^* ; \text{jf}) \setminus \text{cf} = \text{jf} \setminus \text{cf} = \text{jf} \)

**Lemma D.15.** If \( S \) is consistent and promise-free then every extracted execution \( S \triangleright G \) is RC11-consistent.

**Proof.** Let us prove that \( G \) satisfies all the RC11 consistency constraints:

- **RF-NO-THIN-AIR**

  By D.14 we have \( S.\text{po} \cup S.\text{rf} = S.\text{po} \cup S.\text{jf} \). From axiom no-thin-air we know that \( S.\text{po} \cup S.\text{jf} \) is acyclic and therefore \( G.\text{po} \cup G.\text{rf} \subseteq S.\text{po} \cup S.\text{rf} = S.\text{po} \cup S.\text{jf} \) is acyclic as well.

- **COHERENT**

  Follows from the coherence of \( S \) and the fact that \( G.\text{hb} ; G.\text{eco}^2 \subseteq S.\text{hb} ; S.\text{eco}^2 \).

**Lemma D.16.** Given consistent event structure \( S \), if additionally it is promise-free then for every event \( e \in S.\text{E} \) its \((\text{po} \cup \text{jf}) \) prefix is extractable execution graph, i.e., for \( G \triangleq S |_E \), where \( E = \lfloor e \rfloor_{(\text{po} \cup \text{jf})^*} \), we have \( S \triangleright G \).

**Proof.** By definition of justified configuration C.6 and extracted execution C.7 we need to show that \( E \) forms a justified configuration. Since \( E \) is \((\text{po} \cup \text{jf})^* \) downward closure of the event \( e \), then clearly it is \( \text{po} \) downward-closed. \( \text{rf} \) completeness follows from the fact that \( \text{jf} = \text{rf} \) by lemma D.14, \( \text{jf} \) is complete by well-formedness of \( S \), and \( E \) is \((\text{po} \cup \text{jf})^* \) downward closure of the event \( e \). The fact that \( E \) is conflict free follows from the lemma D.12.

**Lemma D.17.** If \( S \) is consistent and promise-free then for every event \( e \) its \((\text{po} \cup \text{jf}) \) prefix forms RC11-consistent execution graph.

**Proof.** Direct consequence of lemmas D.15 and D.16.

**Lemma D.18.** For a consistent event structure \( S \), the following is true

\[
\text{cf} \cap \text{jo} \subseteq \text{rpo}^{-1} ; \text{cf}|_{\text{imm}} ; \text{po}^2
\]
Proof. Let \( \langle w, e \rangle \in \text{cf} \cap \text{jo} \). Then there should exists \( \langle r, r' \rangle \in \text{cf}_{\text{imm}} \), s.t. \( \langle r, w \rangle \in \text{po} \) and \( \langle r', e \rangle \in \text{po} \). By well-formedness \( r, r' \in S.R^{\text{rlx}}_1 \), and \( w \in S.W \). Therefore \( \langle r, w \rangle \in \text{po} \). It is left to show that \( \langle r, w \rangle \in \text{rpo} \). By \text{CERTIFIED} there should exists \( w' \), s.t. \( \langle w, w' \rangle \in \text{ew} \) and \( \langle w', e \rangle \in (\text{po} \cup \text{po}^{-1})^\ast \). Then we get that \( w, w' \in W^{\text{rlx}}_1 \) by well-formedness of \( S \). Next, suppose there is a acquire/release fence \( f \) between \( r \) and \( w \). Then it would be the case that either \( \langle f, w \rangle \in [F];\text{po} \). But since \( \langle r, f \rangle \in \text{po} \) we have \( \langle f, w' \rangle \not\in \text{po} \) which contradicts \text{WELL-FENCED}.

\[ \square \]

Lemma D.19. In the consistent event structure \( S \) promises can only be relaxed:
\[ \text{dom}(\text{cf} \cap \text{jo}) \subseteq W^{\text{rlx}}_1 \]

Proof. Suppose there exists \( w \in \text{dom}(\text{cf} \cap \text{jo}) \), s.t. \( w \in W^{\text{ral}}_1 \). Then by axiom \text{CERTIFIED} we have that there exists \( w' \), s.t. \( \langle w, w' \rangle \in \text{ew} \). But then from well-formedness we can deduce that \( w, w' \in W^{\text{rlx}}_1 \) which contradicts assumption that \( w \) is at least release write.

\[ \square \]

Lemma D.20. Let \( S \) be a consistent event structure and let \( e \in S.E \) be one of its events. Suppose \( e \) depends on a promise \( p \), i.e., \( \langle p, e \rangle \in \text{cf} \cap \text{jo} \). Also, assume that a branch to which \( e \) belongs has a certification write \( c \) for \( p \), i.e., \( \langle e, c \rangle \in \text{po} \) and \( \langle p, c \rangle \in \text{ew} \). Finally, let \( r \) and \( r' \) be read events at which two branches diverge, i.e., \( \langle r, p \rangle \in \text{po} \), \( \langle r', e \rangle \in \text{po} \), and \( \langle r, r' \rangle \in \text{cf}_{\text{imm}} \). Then it has to be that \( \langle p, r' \rangle \in \text{cf} \cap \text{jo} \).

In other words, each promise has to be observed by the neighboring certification branch up to the point of immediate conflict between this certification branch and the branch that issued the promise.

Proof. Note that \( \langle r, p \rangle \in \text{po} \) and \( \langle r, r' \rangle \in \text{cf}_{\text{imm}} \). Therefore, since conflict relation propagates along program order, we have that \( \langle p, r' \rangle \in \text{cf} \). It is left to show that \( \langle p, r' \rangle \in \text{jo} \).

Suppose that \( \langle p, r'' \rangle \not\in \text{jo} \). Let \( r'' \) be the first event in a branch between \( r' \) and \( e \) that observes \( p \):

- \( \langle r', r'' \rangle \in \text{po} \)
- \( \langle r'', e \rangle \in \text{po} \)
- \( \langle p, r'' \rangle \in \text{jo} \)
- \( \langle p, r'' \rangle \not\in \text{jo} \)
- \( \text{po} \)

Also let \( w'' \) be the write from which \( r'' \) reads, i.e., \( \langle p, w'' \rangle \in \text{jo} \) and \( \langle w'', r'' \rangle \in \text{jf} \).

Suppose \( \langle w'', r'' \rangle \in \text{jf}^2 \setminus \text{hb} \). Note that \( \text{jf}^2 \setminus \text{hb} \subseteq (\text{po} \cup \text{jf})^+ \). If the last edge in the \( \text{jf}^2 \setminus \text{hb} \) path from \( w'' \) to \( r'' \) is \( \text{po} \) edge then we would have \( \langle p, r'' \rangle \in \text{jo} \); \( \langle \text{po} \cup \text{jf} \rangle^+ ; \text{po} \subseteq \text{jo} ; \text{po} \). This contradicts our assumption. Therefore the last edge in the \( \text{jf}^2 \setminus \text{hb} \) path has to be \( \text{jfe} \); \( [E^{\text{acq}}] \). But then, by \text{WELL-FENCED} it should be the case that \( \langle w'', p \rangle \in \text{jfe} \); \( [E^{\text{acq}}] ; \text{po} \subseteq (\text{po} \cup \text{jf})^+ \) and since also \( \langle p, w'' \rangle \in \text{jo} \subseteq (\text{po} \cup \text{jf})^+ \) we have a \( (\text{po} \cup \text{jf})^+ \) cycle, which contradicts axiom \text{NO-THIN-AIR}. Therefore \( \langle w'', r'' \rangle \not\in \text{jf}^2 \setminus \text{hb} \). In other words \( r'' \) is justified externally. Thus we have \( \langle w'', p \rangle \in (\text{jf} \setminus \text{jf}^2 ; \text{hb}) ; \text{po} ; \text{ew} \)

Also note that \( \langle r'', p \rangle \not\in \text{lbpat} \), since \( \langle r'', p \rangle \not\in \text{cf}_{\text{imm}} \); \( \text{po} \) (the later is true since we know \( \langle r', p \rangle \in \text{cf}_{\text{imm}} ; \text{po} \) and \( \langle r', r'' \rangle \in \text{po} \)).

Thereby the preconditions of axiom \text{NO-BAIT-AND-SWITCH} apply to \( \langle w'', p \rangle \), and we can conclude that \( \langle w'', p \rangle \in \text{jf} ; \text{po} \subseteq (\text{po} \cup \text{jf})^+ \). But then again, using the fact that \( \langle p, w'' \rangle \in \text{jo} \subseteq (\text{po} \cup \text{jf})^+ \), we arrive at contradiction, because we have a \( (\text{po} \cup \text{jf})^+ \) cycle forbidden by axiom \text{NO-THIN-AIR}.

\[ \square \]

Lemma D.21. Let \( P \) be LB-race-free under RC11. Then every consistent event structure \( S \) of \( P \) is \text{PROMISE-FREE}.

Proof. Let \( R \) be a total extension of \( (\text{po} \cup \text{jf})^+ \) partial order. Let \( \langle e_1, ..., e_n \rangle \) be sequence of events of \( S \), ordered w.r.t. \( R \):

Proc. ACM Program. Lang., Vol. 6, No. OOPSLA2, Article 152. Publication date: October 2022.
• $\forall i \in \{1, \ldots, n\}. \ e_i \in S.E$
• $\forall e \in S.E. \ \exists i. \ e = e_i$
• $\forall j \in \{1, \ldots, n\}. \ i < j \Rightarrow (e_i, e_j) \in R$

Let $S_i \triangleq S_{\{e_0, e_1, \ldots, e_i\}}$. We show by induction on $i$ that conditions stated in the lemma holds for $S_0, S_1, \ldots, S_n = S$. Note that thank to the lemmas D.8 and D.10 all $S_i$ are well-formed and promise-free-consistent.

**Base Case:** $P(i = 0)$.

$S_1.E = \{e_0\}$. In this case $S_1.jo = \emptyset$. Thus **Promise-free** is trivially satisfied.

**Induction Step:** $P(i) \Rightarrow P(i + 1)$.

Note the following holds (below $r_i$ is short for $S.r_i$):

\[
p_0 i+1 = p_0 i \cup p_0 i+1 \cup [e_i+1]
\]
\[
jf i+1 = jf i \cup jf i+1 \cup [e_i+1]
\]
\[
jo i+1 = jo i; ((p_0 i+1 \cup jf i+1) \cup [e_i+1]) \cup jf i+1 \cup [e_i+1] \setminus p_0 i+1
\]

Using these equalities we prove **Promise-free** by case analysis on $e_i+1$.

• $e_i+1 \in S.F$

  In this case $j_1 f i+1 : [e_i+1] = \emptyset$ and thus $jo i+1 = jo i; (p_0 i+1 ; [e_i+1])$. If $e_i+1$ has no $p_0 i+1$-predecessors, then $jo i+1 = jo i$ and thus we have $c_f i+1 \cap jo i+1 = c_f i \cap jo i = \emptyset$. Otherwise we have

\[
c_f i+1 \cap jo i+1 = c_f i \cap jo i \cup \cup c_f i+1 \cap (jo i; p_0 i+1 ; [e_i+1])
\]

We have $c_f i \cap jo i = \emptyset$ by inductive assumption. We prove $c_f i+1 \cap (jo i; p_0 i+1 ; [e_i+1]) = \emptyset$ by contradiction. Let $p \in dom(c_f i+1 \cap (jo i; p_0 i+1 ; [e_i+1]))$. Note that $p \in S.E_i$. Let $e_j$ be immediate $p_0 i+1$-predecessor of $e_i+1$. Then $p \in dom(jo i+1; p_0 i+1 ; [e_i+1])$ implies $p \in dom(jo i; [e_j])$.

Event $e_j$ cannot be in conflict with $p$, since it would violate inductive assumption $c_f i \cap jo i = \emptyset$. Thus either $\langle p, e_j \rangle \in p_0 i$ or $\langle e_j, p \rangle \in p_0 i$. If $\langle p, e_j \rangle \in p_0 i$ then $\langle p, e_i+1 \rangle \in p_0 i+1$, but it contradicts $\langle p, e_i+1 \rangle \in c_f i+1$. If $\langle e_j, p \rangle \in p_0 i$, then since $\langle p, e_j \rangle \in jo i$ by $jo$; $po \subseteq jo$, which follows immediately from the definition of $jo$, we have $\langle p, p \rangle \in jo i$. But $jo i \subseteq (p_0 i \cup jf i) \cup$ and thus we have contradiction with **No-thin-air**. Therefore $c_f i+1 \cap jo i+1 = \emptyset$.

• $e_i+1 \in S.W$

  This case is similar to the previous one.

• $e_i+1 \in S.R$

  In this case we have:

\[
c_f i+1 \cap jo i+1 = c_f i \cap jo i \cup \cup c_f i+1 \cap (jf i+1 ; [e_i+1] \setminus p_0 i+1) \cup \cup c_f i+1 \cap (jo i; p_0 i+1 ; [e_i+1]) \cup \cup c_f i+1 \cap (jo i; jf i+1 ; [e_i+1])
\]

By inductive assumption $c_f i \cap jo i = \emptyset$. By well-formedness $c_f i+1 \cap jf i+1 = \emptyset$ and thus $c_f i+1 \cap (jf i+1 ; [e_i+1] \setminus p_0 i+1) = \emptyset$. $c_f i+1 \cap (jo i; p_0 i+1 ; [e_i+1]) = \emptyset$ by the same argument as in cases $e_i+1 \in S.F$ and $e_i+1 \in S.W$. 

Proc. ACM Program. Lang., Vol. 6, No. OOPSLA2, Article 152. Publication date: October 2022.
We only left to show that \( \text{cf}_{i+1} \cap (\text{jo}_i \cup \text{rf}_{i+1} \cup [e_{i+1}]) = \emptyset \). Let \( e_j \) be immediate \( \text{po}_{i+1} \)-predecessor of \( e_{i+1} \) and let \( p \in \text{dom}(\text{cf}_{i+1} \cap (\text{jo}_i \cup \text{rf}_{i+1} \cup [e_{i+1}])) \). By lemma D.20 the branch of \( e_{i+1} \) should observe promise \( p \) up to the point of immediate conflict. Therefore if \( \langle p, e_j \rangle \in \text{cf}_i \) it should also be true that \( \langle p, e_j \rangle \in \text{jo}_i \). But it would violate the inductive assumption \( \text{cf}_i \cap \text{jo}_i = \emptyset \). Thus \( \langle p, e_j \rangle \notin \text{cf}_i \).

Then either \( \langle p, e_j \rangle \in \text{po}_i \) or \( \langle e_j, p \rangle \in \text{po}_i \). The former cannot be true since then we would have \( \langle p, e_{i+1} \rangle \in \text{po}_{i+1} \), which contradicts \( \langle p, e_{i+1} \rangle \in \text{cf}_{i+1} \). Therefore \( \langle e_j, p \rangle \in \text{po}_i \).

Let \( r \) be an immediate \( \text{po}_i \)-successor of \( e_j \) s.t. \( \langle r, p \rangle \in \text{po}_i \). By definition of immediate conflict we have \( \langle r, e_{i+1} \rangle \in \text{cf}_{i+1} \). By well-formedness we have that \( r \in R^{1x} \) and \( S.\text{loc}(r) = S.\text{loc}(e_{i+1}) \). Also, by lemma D.18 we get \( \langle r, p \rangle \in \text{rpo}_i \). Let \( w \) be a write that justifies \( e_{i+1} \), i.e., \( \langle w, e_{i+1} \rangle \in \text{rf}_{i+1} \). By well-formedness \( S.\text{loc}(w) = S.\text{loc}(e_{i+1}) \). Then \( \langle p, e_{i+1} \rangle \in \text{jo}_{i+1} \cup \text{rf}_{i+1} \cup [e_{i+1}] \), which, in turn, implies \( \langle p, w \rangle \in (\text{po}_i \cup \text{rf}_i)^+ \). Since \( \langle r, p \rangle \in \text{rpo}_i \) we also have \( \langle r, w \rangle \in \text{rpo}_i \cup \text{rf}_i \). Let \( E = \langle w \rangle_{(\text{po}_i \cup \text{rf}_i)^+} \). Note that \( r, w \in E \). Because \( S_i \) is promote-free by D.17 we have that \( G = S_i[E] \) is RC11-consistent execution graph. By D.14 we also have that \( (\text{po}_i \cup \text{rf}_i)^+ = (\text{po}_i \cup \text{rf}_i)^+ \) and thus \( \langle r, w \rangle \in \text{rpo}_i \cup \text{rf}_i \). To show that \( \langle r, w \rangle \) forms a load-buffering race in \( G \) we only have to prove that \( \langle r, w \rangle \notin \text{hb}_i \).

Suppose \( \langle r, w \rangle \in \text{hb}_i \). Since \( \langle r, w \rangle \notin \text{po}_i \), there should exists either.

\begin{itemize}
  \item Write \( w_{r1} \in S.\text{w}[r1] \), s.t. \( \langle r_{r1} \rangle \in \text{po}_i \) and \( \langle w_{r1}, w \rangle \in \text{hb}_i \cup \text{po}_i \). But in this case we have \( \langle w_{r1}, e_{i+1} \rangle \in \text{cf}_{i+1} \cap \text{jo}_{i+1} \), i.e., \( w_{r1} \) is a promise. Then by lemma D.19 \( w_{r1} \in S.\text{w}[r1] \) which contradicts the fact that \( w_{r1} \) is a release write.
  \item Fence \( f_{r1} \in S.\text{f}[r1] \), s.t. \( \langle r_{r1} \rangle \in \text{po}_i \) and \( \langle f_{r1}, w \rangle \in \text{hb}_i \). But in this case \( \langle f_{r1}, p \rangle \in \text{po}_i \) which contradicts the fact that \( \langle r, p \rangle \in \text{rpo}_i \).
\end{itemize}

Therefore the pair \( \langle r, w \rangle \) forms a load-buffering race in RC11-consistent execution graph \( G \), which contradicts our assumption that \( P \) is LB race-free. Hereby, by contradiction we have proved that \( \text{jo}_{i+1} \cup \text{rf}_{i+1} \cup [e_{i+1}] = \emptyset \), which implies \( \text{cf}_{i+1} \cap \text{jo}_{i+1} = \emptyset \).

\( \square \)

**Theorem D.22.** Let \( P \) be LB race-free under RC11. Then the set of its RC11 consistent execution graphs coincides with the set of its Weakestmo2 consistent graphs.

**Proof.** By lemma D.2 every RC11 consistent graph of \( P \) is also Weakestmo consistent.

Let \( G \) be Weakestmo consistent, i.e., there exists Weakestmo consistent event structure \( S \), s.t. \( S \triangleright G \). By lemma D.21 \( S \) is promote-free. Therefore, by lemma D.15, \( G \) is RC11 consistent execution graph of \( P \).

\( \square \)

**E Extending the LBRF Theorem to the Full Weakestmo2 Model**

In this section, we describe how the proof of the LBRF\textsubscript{RC11} theorem can be extended to account for the remaining features of Weakestmo2, namely read-modify-write atomics and sequentially consistent accesses/fences. We observe that there is a certain difficulty in achieving this goal and thus we propose two different solutions to overcome it. We admit that neither solution is ideal and requires a certain trade-off.

But first, let us present the version of RC11 consistency covering all the features of C11.

**Definition E.1.** Execution graph \( G \) is RC11 consistent if the following hold:

- \( \text{po} \cup \text{rf} \) is acyclic \( \text{(RF-NO-THIN-AIR)} \)
- \( \text{hb} \) is irreflexive. \( \text{(COHERENT)} \)
- \( \text{rmw} \cap (\text{fr} \cup \text{co}) = \emptyset \). \( \text{(RMW-ATOMIC)} \)

---

Proc. ACM Program. Lang., Vol. 6, No. OOPSLA2, Article 152. Publication date: October 2022.
Next, note that in the proof of D.21 at some point we have to construct RC11 consistent execution graph from the po ∪ jf prefix of the event structure. There, the fact that po ∪ jf prefix would satisfy RC11 consistency constraints followed almost trivially. In the full model, however, the prefix should also satisfy rmw-atomic and imm-sequential-consistency. There are two options how it can be achieved.

The first obvious option would be to strengthen the Weakestmo2 consistency further and explicitly require from each po ∪ jf conflict-free prefix of the event structure to satisfy these constraints. While this strengthening looks reasonable and harmless, we have observed that at least in the case of rmw-atomic axiom, it has undesirable consequences. The model becomes strictly stronger and the following transformation ends up to be unsound:

\[ a : R_{ex}^{rlx} ; b : W_{ex}^{rlx} ; c : W_{ex}^{rlx} \rightsquigarrow c : W_{ex}^{rlx} ; a : R_{ex}^{rlx} ; b : W_{ex}^{rlx} \]

The second option is to relax the definition of LB race and try to prove that for every LB-race-free program (according to the revised definition) each po ∪ jf conflict-free prefix of the event structure would satisfy rmw-atomic. We managed to do that with the assumption that P has no sc accesses or fences. Below we present the proof.

**Definition E.2.** Given a well-formed execution graph G a pair of events \( \langle e_1, e_2 \rangle \) is called a load buffering race if it satisfies the following:
- \( \text{G.loc}(e_1) = \text{G.loc}(e_2) \)
- \( \langle e_1, e_2 \rangle \notin \text{G.hb} \cup \text{G.hb}^{-1} \)
- Either
  - \( \langle e_1, e_2 \rangle \in [R_{ex}^{rlx}] ; rpo ; (rf \setminus po) ; (po \cup rf)^+ ; [W] \) or
  - \( \langle e_1, e_2 \rangle \in [W_{ex}^{rlx} \cup R_{ex}^{rlx}] ; (po \cup rf)^+ \setminus rf ; [R_{ex}^{rlx}] \)

**Definition E.3.** Event structure S is prefix-rmw-atomic if for every event \( e \in S.E \) its (po ∪ jf) prefix satisfies atomicity, i.e., for \( E = [e]_{(po \cup jf)} \), the following is true:
- \( \text{rmw}_{E} \cap (\text{fr}_{E} \cup \text{co}_{E}) = \emptyset \) (PREFIX-RMW-ATOMIC)

**Definition E.4.** A triple of events \( \langle w, r_{ex}, w_{ex} \rangle \) is an atomicity violation in an execution graph G if
- \( \langle w_{ex}, r_{ex} \rangle \in G.r_{mw}, \)
- \( \langle r_{ex}, w \rangle \in G.fr, \) and
- \( \langle w, w_{ex} \rangle \in G.co. \)

**Definition E.5.** An atomicity violation \( \langle w, r_{ex}, w_{ex} \rangle \) in execution graph G is co-repairable if the following conditions are met:
- \( w \notin G.w_{ex}. \)
- Either w or \( w_{ex} \) is \( (po \cup rf)^+ \) maximal.

**Definition E.6.** Two executions are equivalent modulo \( \text{co} \), written \( G \equiv_{\text{co}} G' \), if they agree on all components except \( \text{co} \), that is:
1. \( G'.E = G.E. \)
2. \( G'.\text{tid} = G.\text{tid} \)
3. \( G'.\text{lab} = G.\text{lab} \)
4. \( G'.\text{po} = G.\text{po} \)
5. \( G'.\text{rf} = G.\text{rf} \)

**Lemma E.7.** Let G be an execution graph satisfying all the RC11-consistency constraints except rmw-atomic. Suppose G contains exactly one atomicity violation, \( \langle w, r_{ex}, w_{ex} \rangle \), and further suppose
that the violation is \textit{co}-repairable. Then there exists \textit{RC11}-consistent execution graph \(G'\) equivalent to \(G\) modulo \textit{co}.

\textbf{Proof.} Let \(G' \triangleq \langle G.E, G.tid, G.1ab, G.po, G.rf, \text{id}, \text{co}' \rangle\) where
\[
\Delta_{\text{co}} = \{\langle w, w_{ex}\rangle\} \quad \text{and} \quad \text{co}' = G.\text{co} \setminus \Delta_{\text{co}} \cup \Delta_{\text{co}}^{-1}
\]
That is, \(\text{co}'\) is the same as \(\text{co}\) except that we swapped a single edge from \(\langle w, w_{ex}\rangle\) to \(\langle w_{ex}, w\rangle\).

Note that \(G'\) trivially satisfies condition \(G' \equiv_{\text{co}} G\).

Let us show that \(\text{co}'\) is well-formed, i.e., it is a total order. We only need to show that irreflexivity since the totality would follow from the totality of \(\text{co}\).

Consider the following inequality:
\[
(G.\text{co} \setminus \Delta_{\text{co}}) \cup (G.\text{co} \setminus \Delta_{\text{co}})^{-1} = (G.\text{co} \setminus \Delta_{\text{co}}) \cup \Delta_{\text{co}}^{-1} \cup (G.\text{co} \setminus \Delta_{\text{co}}) \cup \Delta_{\text{co}}^{-1}
\]
First note that \((G.\text{co} \setminus \Delta_{\text{co}}) \cup (G.\text{co} \setminus \Delta_{\text{co}})^{-1} \subseteq (G.\text{co} \setminus \Delta_{\text{co}}).\) Indeed, suppose there exists \(w'\), s.t. \(\langle w, w'\rangle \in G.\text{co}\) and \(\langle w', w_{ex}\rangle \in G.\text{co}\). But then, since \(\langle w_{ex}, w\rangle \in \text{fr}\) we would also have \(\langle w_{ex}, w'\rangle \in \text{fr}\). Therefore \(\langle w', r_{ex}, w_{ex}\rangle\) would form an atomicity violation in \(G\), which contradicts the assumption that \(\langle w, r_{ex}, w_{ex}\rangle\) is single atomicity violation. For the same reasons \((G.\text{co} \setminus \Delta_{\text{co}}) \cup \Delta_{\text{co}}^{-1} \subseteq G.\text{co} \setminus \Delta_{\text{co}}\) and \(\Delta_{\text{co}}^{-1} \subseteq G.\text{co} \setminus \Delta_{\text{co}}\). Therefore we have:
\[
\text{co}' \triangleq (G.\text{co} \setminus \Delta_{\text{co}} \cup \Delta_{\text{co}}^{-1})^+ = G.\text{co} \setminus \Delta_{\text{co}} \cup \Delta_{\text{co}}^{-1}
\]
Note that irreflexivity of relation \(r\) is equivalent to \(r \cap \text{id} = \emptyset\) and thus irreflexivity of the union of relations is equivalent to irreflexivity of components. Irreflexivity \(\text{co} \setminus \Delta_{\text{co}}\) follows from irreflexivity of \(\text{co}\), and \(\Delta_{\text{co}}^{-1}\) is trivially irreflexive. Therefore \(\text{co}'\) is well-formed and thus \(G'\) is well-formed as well.

Since \(G'.po = G.po, G'.rf = G.rf,\) and \(G\) satisfies \textit{RF-NO-THIN-AIR} we have that \(G'\) also satisfies \textit{RF-NO-THIN-AIR}.

Because \(\langle w, r_{ex}, w_{ex}\rangle\) is a single atomicity violation in \(G\) and it is clearly not an atomicity violation in \(G'\) (because \(\langle w, w_{ex}\rangle \notin \text{co}'\) as was shown above) then \(G'\) satisfies \textit{RMW-ATOMIC}.

Also note that since \(\text{hb}\) is defined in terms of \textit{1ab, po, and rf} we have that \(G'.\text{hb} = G.\text{hb}\). Therefore \(G.\text{hb}'\) is irreflexive.

We need to show \(G'.\text{hb} ; G'.\text{eco}\) is irreflexive.

First, note that for any well-formed execution graph (or, more general, event structure) the following equation can be proven:
\[
\text{eco} \triangleq (\text{rf} \cup \text{co} \cup \text{fr})^+ = \text{rf} \cup \text{co} ; \text{fr}^2 \cup \text{fr}^2 ; \text{fr}^2
\]
Then observe that:
\[
G'.\text{eco} = G.\text{rf} \cup G'.\text{co} ; G.\text{rf}^2 \cup G.\text{rf}^{-1} ; G'.\text{co} ; G.\text{rf}^2
\]
Irreflexivity of the following components follows from the fact that \(G\) is \textit{COHERENT}.

- \(G.\text{hb} ; G.\text{rf}\) is irreflexive.
Then it would imply

\[ \langle e, r_{\text{ex}} \rangle \in hb \]

Thus adding the assumption that \( G \) is COHERENT.

Next, suppose that \( \langle w, w_{\text{ex}} \rangle \in G \cdot rf \). By our assumption either \( w \) or \( w_{\text{ex}} \) is \((G.p_o \cup G.rf)^* \) maximal. If \( w_{\text{ex}} \) is \((G.p_o \cup G.rf)^* \) maximal then \( G \cdot rf \) is maximal. Similarly, \( [w] \) is maximal. Thus we have only left to consider the case \( \langle w, w_{\text{ex}} \rangle \in G.hb \) and \( w \) is \((G.p_o \cup G.rf)^* \) maximal. But then notice that \( G.hb \subseteq (G.p_o \cup G.rf)^* \) and thus \([w] ; G.hb \) is empty as well.

**Definition E.8.** An atomicity violation \( \langle w'_{\text{ex}}, r_{\text{ex}}, w_{\text{ex}} \rangle \) in execution graph \( G \) is **rf-repairable** if:

- \( w'_{\text{ex}} \in G.w_{\text{ex}} \) and
- \( w_{\text{ex}} \) is \( p_o \cup rf \) maximal.

**Lemma E.9.** Let \( G \) be an execution graph satisfying all the RC11-consistency constraints except \( \text{RMW-ATOMIC} \). Let \( \langle w'_{\text{ex}}, r_{\text{ex}}, w_{\text{ex}} \rangle \) be a single atomicity violation and let it be a rf-repairable. Then there exists RC11 consistent execution graph \( G' \) equivalent to \( G \) modulo \( rf \) \((w'_{\text{ex}}, r_{\text{ex}})\), denoted as \( G' \equiv rf(w'_{\text{ex}}, r_{\text{ex}}) \) \( G \) in the following sense:

1. \( G'.E = G.E \setminus \{w_{\text{ex}}, r_{\text{ex}}\} \cup \{r'_{\text{ex}}\} \) where:
   - \( r'_{\text{ex}} = \langle G.\text{id}(r_{\text{ex}}), G.\text{tid}(r_{\text{ex}}), lab \rangle \)
   - \( lab = R_{\text{ex}}(l, v) \)
   - \( o = G.\text{mod}(r_{\text{ex}}) \)
   - \( l = G.\text{loc}(r_{\text{ex}}) \)
   - \( v = G.\text{val}(w_{\text{ex}}) \)

2. \( G'.p_o = G.p_o \setminus G.E' \).
3. \( G'.rf = G.rf \setminus \{r_{\text{ex}}\} \cup \Delta_{rf} \) where \( \Delta_{rf} = \{\langle w'_{\text{ex}}, r'_{\text{ex}} \rangle\} \)
4. \( G'.c_o = G.c_o \setminus G.E' \).

**Proof.** It is easy to show that \( G' \) is well-formed.

Since \( w_{\text{ex}} \) is \( p_o \cup rf \) maximal in \( G \) then \( r_{\text{ex}} \) is \( p_o \cup rf \) maximal in \( G' \). Thus adding \( \Delta_{rf} \) cannot introduce any cycle in \( G.p_o \cup G.rf \) and therefore \( G'.p_o \cup G'.rf \) is acyclic, i.e., \( G' \) sat. \( \text{RF-NO-THIN-AIR} \).

The fact that \( G' \) is **RMW-ATOMIC** follows from assumption that \((w'_{\text{ex}}, r_{\text{ex}}, w_{\text{ex}})\) is the only atomicity violation in \( G \). Since we have removed \( w_{\text{ex}} \) from the graph \( G' \) has no other atomicity violations.

Thus we only left to show that \( G'.hb ; G'.eco \) is irreflexive.

- \( G'.hb \) is irreflexive since \( G.hb \subseteq (G.p_o \cup G.rf)^* \).
- Irreflexivity of \( G'.hb \); \( G'.rf \) follows from irreflexivity of \( G.hb \); \( G.rf \) and the fact that \( r_{\text{ex}} \) is \((G.p_o \cup G'.rf)^* \) maximal.
- Irreflexivity of \( G'.hb \); \( G'.c_o = G.hb \); \( G.c_o = G.hb \); \( G.c_o \) follows directly.
- Let us consider \( G'.hb \); \( G'.c_o \); \( G'.rf \).

\[
G'.hb ; G'.c_o ; G'.rf = G.hb ; G.c_o ; G'.rf
\]

But the cycle \( \langle r_{\text{ex}}', r_{\text{ex}}' \rangle \in G.hb ; G.c_o ; \Delta_{rf} \) would contradict \( G'.p_o \cup G'.rf \) maximality of \( r_{\text{ex}} \).

Finally, suppose there is cycle \( \langle r'_{\text{ex}}', r'_{\text{ex}}' \rangle \in G'.fr ; G'.rf \). Let \( w \) be the endpoint of the first edge in the cycle, i.e., \( (r_{\text{ex}}', w) \in G'.fr \). Then we would have \((w', w) \in G.c_o \) and therefore \( \langle w, r_{\text{ex}} \rangle \in G.fr ; G.rf \); \( G.hb \) would also close a cycle in \( G.hb ; G.eco \).
Finally, we need to adjust the proof of the lemma D.21. Namely, along proving by induction that each $S_i$ is PROMISE-FREE, we also need to prove that it is PREFIX-RMW-ATOMIC.

**Proof.** The base of the induction is trivial again, since $S_0.RMW = \emptyset$. Thus we only consider the step of the induction. Note the following holds:

\[
RMW_{i+1} = RMW_i \cup RMW_{i+1} \cap [e_{i+1}]
\]

\[
RMW_{i+1} \cap (fr_{i+1} ; co_{i+1}) = \\
= RMW_i \cap (fr_i ; co_i)
\]

\[
\cup RMW_i \cap (fr_i ; [e_{i+1}] ; co_i)
\]

\[
\cup (RMW_{i+1} \cap [e_{i+1}]) \cap (fr_i ; co_{i+1} ; [e_{i+1}])
\]

Using these equalities we prove PREFIX-RMW-ATOMIC by case analysis on $e_{i+1}$.

- **$e_{i+1} \in S.F$**
  
  In this case $fr_{i+1} ; [e_{i+1}] = \emptyset$, $co_{i+1} ; [e_{i+1}] = \emptyset$, and $RMW_{i+1} \cap [e_{i+1}] = \emptyset$. Thus $RMW_{i+1} \cap (fr_{i+1} ; co_{i+1}) = RMW_i \cap (fr_i ; co_i) = \emptyset$. Therefore, for every event $e \in S.E_{i+1}, E = [e]_{(po \cup jf)}$, the required property $RMW_{i+1} \cap (fr_{i+1} ; co_{i+1}) = \emptyset$ follows immediately.

- **$e_{i+1} \in S.R$**
  
  This case is similar to the previous one.

- **$e_{i+1} \in S.W$**
  
  For every $e \in S.E_i$ the rmw-atomicity of its prefix follows immediately from the inductive assumption. Thus we only need to show that prefix of $e_{i+1}$ satisfies rmw-atomicity. Let $E = [e]_{(po \cup jf)}$. We need to show:

\[
RMW_{i+1} | E \cap (fr_{i+1} | E ; co_{i+1} | E) = \\
= RMW_i | E \cap (fr_i | E ; co_i | E)
\]

\[
\cup RMW_i | E \cap (fr_i | E ; [e_{i+1}] ; co_i | E)
\]

\[
\cup (RMW_{i+1} \cap [e_{i+1}]) \cap (fr_i | E ; co_{i+1} | E ; [e_{i+1}])
\]

- $RMW_i | E \cap (fr_i | E ; co_i | E) = \emptyset$ by inductive assumption.

- Let $\langle r_{ex} , w_{ex} \rangle \in RMW_i | E$, s.t. $\langle r_{ex} , e_{i+1} \rangle \in fr_{i+1}$ and $\langle e_{i+1} , w_{ex} \rangle \in co_{i+1}$. Note that by well-formedness we have that $r_{ex} \in S_i.R_{ex}$ and we have $\langle r_{ex} , e_{i+1} \rangle , \langle w_{ex} , e_{i+1} \rangle \in (po \cup jf)^\ast$. Consider the cases for $e_{i+1}$.

  - **$e_{i+1} \in S.W_{ex}$**
    
    Let $e_j$ be immediate po predecessor if $e_{i+1}$. Then by well-formedness $\langle e_j , e_{i+1} \rangle \in S_{i+1}.RMW$. Then $\langle w_{ex} , e_j , e_{i+1} \rangle$ forms rf-repairable atomicity violation in $E$. Note that from inductive assumption it follows that it is the only atomicity violation in $E$. Then, giving that $S_{i+1}$ is PROMISE-FREE, using the lemmas D.17 and E.5, we can conclude that there exists RC1 consistent execution graph $G$, s.t. $G \equiv_{rf(e_j)} S_{i+1} | E$. We have $\langle r_{ex} , e_j \rangle \in [R_{ex}] ; (G.po \cup G.rf)^\ast ; [R_{ex}].G$, i.e., the pair $\langle r_{ex} , e_j \rangle$ forms load-buffering race in RC11-consistent execution $G$, which contradicts our assumption that $E$ is LB-race-free.

  - **$e_{i+1} \notin S.W_{ex}$**
    
    Then $\langle e_{i+1} , r_{ex} , w_{ex} \rangle$ forms a co-repairable atomicity violation in $E$. Note that from inductive assumption it follows that it is the only atomicity violation in $E$. Then, giving that $S_{i+1}$ is PROMISE-FREE, using the lemmas D.17 and E.5, we can conclude that there exists RC11 consistent execution graph $G$, s.t. $G \equiv_{co} S_{i+1} | E$. Since $S_{i+1}$ is PROMISE-FREE, by D.14
we also have that

\[(p_{0|t+1} \cup j_{f|t+1})^+ = (p_{0|t+1} \cup r_{f|t+1})^+ = (G.p_o \cup G.r_f)^+\]

Therefore the pair \([r_{ex}, e_{|t+1}] \in [R_{ex}] : (G.p_o \cup G.r_f)^+ ; [R_{ex}]\) forms a load-buffering race in RC11-consistent execution graph \(G\), which contradicts our assumption that \(P\) is LB-race-free.

- Let \((e_j, e_{|t+1}) \in R_{wm|t+1}\). Let \(w \in E\), s.t. \((e_j, w) \in f_{rf|t}\) and \((w, e_{|t+1}) \in c_{co|t+1}\). Then similarly to the previous cases, we can show that there exists a RC11-consistent execution graph \(G\) of program \(P\), that contains a load-buffering race.

\[\square\]

F CORRECTNESS OF COMPILATION MAPPINGS FOR WEAKESTMO2

In this section, we present the proof of the correctness of compilation from Weakestmo2 to IMM.

**Theorem F.1.** Let \(P\) be a program, and \(G\) be an IMM-consistent execution graph of \(P\). Then, there exists a Weakestmo2-consistent event structure \(S\) of \(P\) such that \(S \triangleright G\).

Our development is based on the proof of the same statement for the original Weakestmo model by Moiseenko et al. [2020].

**F.1 Recap of the Original Proof Structure**

Moiseenko et al. [2020] construct the required event structure \(S\) step by step following a traversal of the IMM graph \(G\) [Moiseenko et al. 2020, §2.3]. Traversal of the graph \(G\) induces operational small-step semantics \(G \vdash TC \xrightarrow{SA} TC'\) where \(TC\) and \(TC'\) are traversal configurations and \(e\) is an event being traversed. A traversal configuration is a tuple \((C, I)\), where \(C \subseteq G.E\) is a set of covered events and \(I \subseteq G.W\) is a set of issued writes.

Giving the operational semantics of traversal \(G \vdash TC \xrightarrow{e} TC'\), such that \(G.tid(e) = t\), and the operational semantics of event structure construction \(S \xrightarrow{F} S'\) the proof then proceeds using the standard simulation argument. Moiseenko et al. [2020, §4.2] define a simulation relation \(I(P, T, G, TC, S, X)\) between the program \(P\), a set of thread identifiers \(T \subseteq Tid\), the current traversal configuration \(TC\) of execution graph \(G\), the current Weakestmo-consistent event structure \(S\), and a justified configuration \(X\) of \(S\). They then prove the following three lemmas which state that (i) the initial traversal configuration and initial event structure are related by the simulation relation, (ii) each traversal step can be simulated by an event structure construction step, and (iii) it is possible to extract the traversed execution graph from an event structure corresponding to the terminal traversal configuration.

**Lemma F.2 (Simulation Start).** Let \(P\) be a program, and \(G\) be an IMM-consistent execution graph of \(P\). Then \(I(P, tid(P), G, T_{init}(G), S_{init}(P), \{e_0\})\) holds where

- \(tid(P)\) the set of threads of program \(P\);
- \(T_{init}(G) \triangleq \{e_0, e_0\}\) is initial traversal configuration containing only the initialization event;
- \(S_{init}(P)\) is initial event structure also containing only the initialization event.

**Lemma F.3 (Weak Simulation Step).** If \(I(P, T, G, TC, S, X)\) and \(G \vdash TC \rightarrow TC'\) hold, then there exist \(S'\) and \(X'\) such that \(I(P, T, G, TC', S', X')\) and \(S \rightarrow^* S'\) hold.

**Lemma F.4 (Simulation End).** If \(I(P, tid(P), G, T_{final}(G), S, X)\) holds, where \(T_{final}(G) \triangleq \langle G.E, G.E \rangle\) is a terminal traversal configuration, then the execution graph associated with \(X\) is isomorphic to \(G\), or in other words, \(G\) can be extracted from \(S\): \(S \triangleright G\).
Lemma F.3 is the most challenging lemma to prove. Simulating a traversal step might require to construct a new certification branch $Br_{cert}$ in the event structure containing multiple events [Moiseenko et al. 2020, §4.3]. To construct this branch, one must (i) pick a suitable justification write for each read event [Moiseenko et al. 2020, §4.3.1], and (ii) chose the position of each write event in the event structure’s coherence order [Moiseenko et al. 2020, §4.3.2]. Once the certification branch $Br_{cert}$ is constructed, it is used to replace the events of the corresponding thread in the configuration $X$.  

$$X' \triangleq X \setminus Br_{prev} \cup Br_{cert}$$

$$Br_{prev} \triangleq X|_t \triangleq \{ e \in X \mid S.tid(e) = t \}$$

The new constraint NO-BAIT-AND-SWITCH added to Weakestmo2 applies exactly to the read events in two adjacent branches of an event structure. Thus, in order to adjust the proof for Weakestmo2 model, we need to show that the justification writes chosen for reads in a certification branch match those justification writes in the previous branch that issued the promises.

### F.2 Simulation Relation

Before going into the details of the modified proof for Weakestmo2, we highlight some invariants guaranteed by the simulation relation $I$. We present here only some of invariants derived from $I$ rather than full simulation relation for two reasons. First, the presented invariants should help to develop some intuition about the construction without going into too much technical details. Second, the proof adaptation requires only this subset of simulation relation’s properties. Curious reader may find the full simulation relation in [Moiseenko et al. 2020, §A] or in the Coq developments accompanying that paper [https://github.com/weakmemory/weakestmoToImm/].

The simulation relation $I (P,T,G,TC,S,X)$ establishes a connection between the event structure $S$ and execution graph $G$ with the help of a function $s2g : S.E \rightarrow G.E$ which maps an event of $S$ to an event of $G$. This function can be lifted to sets of events in the following way\(^1\):

for $A_S \subseteq S.E : \llbracket A_S \rrbracket \triangleq \{ s2g(e) \in G.E \mid e \in A_S \}$

for $A_G \subseteq G.E : \llbracket A_G \rrbracket \triangleq \{ e \in S.E \mid s2g(e) \in A_G \}$

The simulation relation $I$ contains the following properties.

1. Events of $S$ restricted to threads $T$ and events of $X$ correspond to covered and issued events and their po-predecessors:
   - $\llbracket S.E_I \rrbracket = \llbracket X \rrbracket = C \cup dom(G.po) ; [I]$

2. Labels of events in $S$ match labels of events in $G$ modulo their values.
   a. $\forall e \in S.E$. $S.\{ tid, typ, loc, mod \}(e) = G.\{ tid, typ, loc, mod \}(\llbracket e \rrbracket)$
   b. $\forall e \in X \cap \llbracket G.D(C,I) \rrbracket$. $S.val(e) = G.val(\llbracket e \rrbracket)$

3. Program order in $S$ corresponds to program order in $G$:
   - $\llbracket S.po \rrbracket \subseteq G.po$

4. Identity relation in $G$ corresponds to identity or conflict relation in $S$:
   - $\llbracket id \rrbracket \subseteq S.cf$

5. Reads in $S$ are justified by writes that have already been observed by the corresponding events in $G$.
   a. $\llbracket S.jf \rrbracket \subseteq G.rf \cup G.hb$

Moreover, restriction of justified-from relation on $X$ corresponds to stable justification relation (see F.3) of the current traversal configuration in graph.

\(^1\)In a similar manner it is possible to lift $s2g$ to binary relations on events.

Proc. ACM Program. Lang., Vol. 6, No. OOPSLA2, Article 152. Publication date: October 2022.
As a consequence it is possible to derive that justification for covered events in $X$ corresponds to their justification in $G$: $[[S.jf ; [X \cap \|C\|]]] \subseteq G.rf$.

Finally, only issued events can be used for external justification.

(c) $dom(S.jf) \subseteq dom(S.ew^* ; [X \cap \|I\|])$

(6) Equivalent writes in $S$ are mapped to the same write in $G$.

(a) $[[S.ew]] \subseteq id$

Also, every $S.ew^*$ equivalence class has a representative issued event in $X$.

(b) $S.ew^* \subseteq (S.ew^* ; [X \cap \|I\|] ; S.ew^*)$

(7) Coherence edges in $S$ correspond to coherence or identity edges in $G$.

(a) $[[S.co]] \subseteq G.co$

Coherence edges in $S$ ending in $X$ restricted to threads $T$ correspond precisely to coherence edges in $G$.

(b) $[[S.co ; [X_T]]] \subseteq G.co$

(8) Synchronize-with and happens-before relations in $S$ conform to the corresponding relations in $G$.

(a) $[[S.sw]] \subseteq G.sw$

(b) $[[S hb]] \subseteq G.hb$

Let us summarize the description of the simulation relation with an informal explanation of its invariants. As we have mentioned, traversal of the execution graph captures a possible execution order of the program leading to the given graph. Covering an event corresponds to execution an instruction of the program in-order, while issuing a write corresponds to out-of-order execution.

The construction of the event structure reflects this execution strategy, with a notable difference being that in event structure each write event executed out-of-order must come together with a sequence of events that lead to this write. The first property of the simulation relation formalizes this intuition. The receptiveness property [Podkopaev et al. 2019, §6.4] guarantees that it is always possible to construct an execution branch leading to a specific issued event, and that this branch will contain the same set of events with almost the same labels, except that values of some events might be different. Hence we have the properties 2 and 3. The property 4 further guarantees that events in the event structure which have the same image in graph $G$ are either equal or in conflict.

The next few properties constrain the justification of reads. First, property 5a ensures that a read event in the event structure cannot be justified from a write located happens-before after the write this event reads-from in the graph. Property 5b asserts that justification of reads in the configuration $X$ should match the stable justification relation in $G$. We postpone the detailed description of this relation until F.3. Property 5c assures that only issued events can be used to justify a read externally, i.e., by a write from another thread. This implies, in particular, that promises in the event structure correspond to issued events in the graph.

Properties 6a and 6b state that each equivalence class of writes in $S$ correspond exactly to some issued write event in $G$. Finally, properties 7, 8a, and 8b relate $S.co$, $S.sw$, and $S.hb$ to their counterparts in the graph $G$.

To further develop intuition about the simulation process we recommend to consult an example in [Moiseenko et al. 2020, §2.4].

Remark. Note that in property 2b we use determined events $G.D_{(C,I)}$ (to be defined later in F.3), while Moiseenko et al. [2020] uses covered and issued events. As will become evident later $C \cup I \subseteq G.D_{(C,I)}$ and thus our property is stronger. Nevertheless, the construction of Moiseenko et al. [2020] actually allows to establish this stronger property, as we will see later. It just that for
the proof for Weakestmo model the weaker property was sufficient, while our modified proof will require the stronger property.

### F.3 Stable Justification

To simulate a traversal step $G \vdash TC \xrightarrow{e} TC'$ of a thread $G.\text{tid}(e)$ that has outstanding promises (i.e., issued writes which are not covered yet), the event structure must take multiple steps to construct a certification branch. For each read in this branch, we must select a write event justifying this read. For this purpose, Moiseenko et al. [2020, §4.3.1] define the *stable justification relation* in a few steps.

First, *determined* events, $G.D_{(C,I)}$, are events that must have the same label in $G$ and in configuration $X$ of $S$.

\[
G.D_{(C,I)} = C \cup I \cup \\
G.W \setminus \text{rng}(G.\text{ppo}) \cup \\
\cup \text{dom}(G.\text{rfi}^2 \cup G.\text{ppo} \cup I) \cup \\
\cup \text{rng}(I \cup G.\text{rfi}) \cup \\
\cup \text{rng}(G.\text{rf} \cup [G.E^\exists])
\]

Next, the *viewfront relation*, $G.\text{vf}_{TC}$, represents which write events have been observed by other events.

\[
G.\text{vf}_{(C,I)} = [G.W] \cup (G.\text{rf} \cup [C])^2 \cup G.\text{hb} \cup G.\text{rf} \cup [G.D_{(C,I)}] \cup G.\text{po}^2
\]

Finally, *stable justification* picks co-maximal observed write.

\[
G.\text{sjf}_{TC} = ([G.W] \cup (G.\text{vf}_{TC} \cap =_{G.1\text{oc}}) \cup [G.R]) \setminus (G.co \cup G.\text{vf}_{TC})
\]

**Remark.** We have slightly modified the definition of determined events compared to the one in [Moiseenko et al. 2020, §4.3.1]. The modified version of the definition will help us later simplify our proofs. The original definition of determined events is shown below.

\[
G.D_{(C,I)} = C \cup I \cup \text{dom}(G.\text{rfi}^2 \cup G.\text{ppo} \cup I) \cup \text{rng}(I \cup G.\text{rfi}) \cup \text{rng}(G.\text{rf} \cup [G.E^\exists])
\]

The only difference of the modified version is that it also includes $G.W \setminus \text{rng}(G.\text{ppo})$, i.e., writes that do no depend on preceding read events, and so their values can be fixed in advance. We argue that this modification does not affect the proofs. The viewfront relation (and thus the stable justification relation too) in fact depends only on the set of determined reads. Making more write events determined does not affect it.

Viewfront and stable justification relations possess a number of useful properties.

**Proposition F.5.** $G.\text{vf}_{TC}; G.\text{eco}$ is irreflexive.

**Proposition F.6.** $G.\text{sjf}_{TC} = G.\text{rfi}$

**Proposition F.7.** $G.\text{sjf}_{TC}; [G.D_{TC}] = G.\text{rf}; [G.D_{TC}]$ In particular, it implies that given a traversal step $G \vdash TC \rightarrow TC'$, we have that:

\[
G.\text{sjf}_{TC}; [G.D_{TC}] = G.\text{rf}; [G.D_{TC}] = G.\text{sjf}_{TC}; [G.D_{TC}]
\]

**Proposition F.8.** Given a traversal step $G \vdash (C,I) \rightarrow (C',I')$, we have that

\[
G.\text{sjf}_{(C',I')} = [I]; G.\text{sjf}_{(C',I')} \cup G.\text{sjf}_{(C,I)}
\]

Put simply, each stable external justification write is issued.

**Proposition F.9.** Given a traversal step $G \vdash (C,I) \rightarrow (C',I')$, we have that

\[
G.\text{sjf}_{(C',I')} \cup [G.E \setminus G.E_t] = G.\text{sjf}_{(C,I)} \cup [G.E \setminus G.E_t]
\]
F.4 Simulation Relation for Certification Branch

Along with the simulation relation \( I(P, T, G, TC, S, X) \), the authors also define an auxiliary relation \( I_{\text{cert}}(P, T, G, TC', S, X, Br_{\text{cert}}) \), which we will call certification simulation relation. This relation holds during the construction of the certification branch [Moiseenko et al. 2020, §A]. It is also used at the end of the construction of the certification branch in order to restore the main simulation relation, i.e., to show that \( I(P, T, G, TC', S', X') \) holds. Below we list core invariants of \( I_{\text{cert}}(P, T, G, (C, I), (C', I'), S, X, Br_{\text{cert}}) \). As in case of the main simulation relation we mention only a subset of properties which would be relevant to our modification of the proof.

1. \( I(P, T \setminus t, G, (C, I), S, X) \) holds.
2. \( G \vdash (C, I) \rightarrow_t (C', I') \), i.e., there is a traversal step, covering/issuing an event from thread \( t \).
3. Certification branch is equal to a branch of the thread \( t \) in the configuration \( X \) up to a first uncovered event.
   a. \( Br_{\text{cert}} \cap \llbracket C \rrbracket = X|_t \cap \llbracket C \rrbracket \).
   In general, set of events of the certification branch is a subset of covered/issued events and their \( \text{po} \)-predecessors from the new traversal configuration.
   b. \( \llbracket Br_{\text{cert}} \rrbracket \subseteq C' \cup \text{dom}(G.\text{po}^2; \llbracket I' \rrbracket) \).
4. Labels of determined events in \( Br_{\text{cert}} \) match labels of events in \( G \).
   a. \( \forall e \in Br_{\text{cert}} \cap \llbracket G.D((C', I')) \rrbracket. S.\text{val}(e) = G.\text{val} (\llbracket e \rrbracket) \)
5. Reads in \( Br_{\text{cert}} \) are justified according to the stable justification relation.
   a. \( \llbracket S.jf ; [Br_{\text{cert}}] \rrbracket \subseteq G.sjf_{TC'} \)
6. For every issued event in \( Br_{\text{cert}} \) there exists \( S.\text{ew} \) equivalent event in \( X \).
   a. \( Br \cap \llbracket I \rrbracket \subseteq \text{dom}(S.\text{ew}^2; \llbracket X \rrbracket) \)
   Symmetrically, every issued event in \( X \) within the processed part of the certification branch has an \( S.\text{ew} \) equivalent event in \( Br_{\text{cert}} \).
   b. \( X \cap \llbracket I \cap \llbracket Br_{\text{cert}} \rrbracket \rrbracket \subseteq \text{dom}(S.\text{ew}^2; [Br_{\text{cert}}]) \)
7. The \( S.\text{co} \) edges ending in \( Br_{\text{cert}} \) corresponds to \( G.\text{co} \) edges.
   a. \( \llbracket S.\text{co} ; [Br_{\text{cert}}] \rrbracket \subseteq G.\text{co} \)
   The \( S.\text{co} \) edges ending in \( X|_t \) and not in the processed part of the certification branch correspond to \( G.\text{co} \) edges.
   b. \( \llbracket S.\text{co} ; [X|_t \setminus \llbracket Br_{\text{cert}} \rrbracket \rrbracket \rrbracket \rrbracket \subseteq G.\text{co} \)

F.5 Adapting the Proof

We now have all the ingredients to adapt the proof for Weakestmo2. Recall the NO-BAIT-AND-SWITCH axiom.

\[
(jf \setminus (jf^2 ; hb)) ; \text{po} ; \text{ew} \subseteq jf ; (\text{po} \cup \text{lpat})
\]

Essentially, we need to show that if the simulation relation \( I(P, T, G, TC, S, X) \) holds, the event structure \( S \) satisfies NO-BAIT-AND-SWITCH, and the traversal makes a step \( G \vdash TC \rightarrow TC' \), then the event structure can take multiple steps \( S \rightarrow^* S' \) to construct a branch \( Br_{\text{cert}} \), such that \( I(P, T, G, TC', S', X') \) holds and \( S' \) also satisfies NO-BAIT-AND-SWITCH.

Moiseenko et al. [2020, §4.3.3] describes how to construct a single branch \( Br_{\text{cert}} \) and \( S' \) such that \( I(P, T, G, TC', S', X') \) holds, where \( S' \) is equal to original event structure \( S \) extended with the new events from \( Br_{\text{cert}} \). The branch \( Br_{\text{cert}} \) is built with the help of the receptiveness property using \( G.sjf_{TC'} \) to chose justification writes for read events. Unfortunately, if we take just this branch \( Br_{\text{cert}} \), then \( S' \) might actually violate NO-BAIT-AND-SWITCH.
In order to fix the proof, we need to consider a series of certification branches \(\{Br_1, ..., Br_{n+1} = Br_{\text{cert}}\}\), where each branch is guided by a variant of the stable justification relation \(G.sjf_{TC}\) (to be defined later). We can then show that each corresponding event structure \(S'\) satisfies NO-BAIT-AND-SWITCH and a weaker version of the simulation relation \(I_{\text{weak}}^i(P, G, TC, TC', S', X_i)\). We then demonstrate that the final event structure \(S_{n+1}'\) in this series satisfies the main simulation relation \(I(P, T, G, TC, S_{n+1}', X_{n+1})\). The weak version of the simulation relation \(I_{\text{weak}}^i(P, T, G, TC, TC', S, X)\) is parameterized by the natural number \(i\) and two consecutive traversal configurations \(G \vdash TC \rightarrow TC'\). It has the following differences comparing to the regular simulation relation \(I(P, T, G, TC, S, X)\).

- In 2b instead of \(G.D_{(C, I)}\) we take \(G.D_{(C', P')}\).
- \(\forall e \in X \cap \|G.D_{(C', P')}\|\). \(S.val(e) = G.val(\|e\|)\)
- In 5b instead of \(G.sjf_{TC}\) we take \(G.sjf_{TC'}\).

We first briefly explain the intuition behind the construction of multiple branches. The newly issued write \(w\) might depend on several read events via the \(G.ppo\) order. It means that the presence of \(w\) in certification branch and its label depend on the labels of these reads. The definition of \(G.sjf_{TC'}\) ensures that event \(w\) would appear in certification branch and would have suitable label by choosing appropriate justification writes for all reads on which \(w\) depends via \(G.ppo\). Changing justification for multiple reads at once violates NO-BAIT-AND-SWITCH, because the previous branch \(Br_{\text{prev}}\) of \(S\) and the certification branch \(Br_{\text{cert}}\) would have different set of incoming justification writes. It is possible, however, to replace justification writes for \(G.ppo\)-preceding reads incrementally one-by-one, constructing a series of certification branches \(\{Br_1, ..., Br_{n+1}\}\). This way, any two consecutive branches in this sequence would only disagree on a single justification write located at the point of their immediate conflict, a situation which is allowed by axiom NO-BAIT-AND-SWITCH with the help of the load-buffering pattern relation \(\text{lbpat}\). Note that the newly constructed branches do not alter justification writes for reads \(G.ppo\)-preceding previously issued writes \(I\), because \(\text{dom}(G.ppo ; [I]) \subseteq G.D_{(C, I)}\) and \(G.sjf_{TC'} ; [G.D_{TC}] = G.sjf_{TC} ; [G.D_{TC}]\). In other words, all branches \(\{Br_1, ..., Br_{n+1}\}\) would contain writes equal to writes from \(I\), which would allow us to certify all pending promises in each branch (remember that issued events correspond to promises).

Next we present a more formal version of the proof explained above. We would need several auxiliary definitions, propositions, and lemmas.

**Lemma F.10.** The set of determined reads can be characterized as follows.

\[
G.R \cap G.D_{(C, I)} = G.R \cap C \cup \text{dom}(G.ppo ; [I]) \cup \text{rng}([I] ; G.rf) \cup \text{rng}(G.ref ; [G.E_{\text{acq}}])
\]

**Proof.** Follows directly from the definition of determined events and the following facts:

- \(G.R \cap G.W = \emptyset\);
- \(I \subseteq G.W\);
- \(\text{dom}(G.rf) \subseteq G.W\).

**Lemma F.11.** The viewfront relation can be equivalently defined as follows.

\[
G.vf_{(C, I)} = [G.W] ; (G.rf ; [C])^2 ; G.hb^2 \cup G.ref \cup \text{dom}(G.ppo ; [I]) ; G.po^2 \cup G.ref ; [G.E_{\text{acq}}] ; G.po^2
\]

**Proof.** Note that \(\text{rng}(G.rf) \subseteq R\) and thus \(G.rf ; [G.D_{(C, I)}] = G.rf ; [R \cap G.D_{(C, I)}]\). Using Lemma F.10 we can rewrite \(G.vf_{(C, I)}\) as follows.

\[
G.vf_{(C, I)} = [G.W] ; (G.rf ; [C])^2 ; G.hb^2 \cup
\]

Proc. ACM Program. Lang., Vol. 6, No. OOPSLA2, Article 152. Publication date: October 2022.
Applying the following (in)equalities we arrive to the conclusion.

\[ G.D \subseteq [G.W \cup (G.rf ; [C])^2 \cup G.po^2 \cup \]
\[ G.rf ; [dom(G.ppo ; [I])] \cup G.po^2 \cup \]
\[ G.rf ; [rng([I] ; G.rfi)] \cup G.po^2 \cup \]
\[ G.rf ; [rng(G.rfe ; [G.E^{2acq}])] \cup G.po^2 \]

G.rf ; [rng([I] ; G.rfi)] \cup G.po^2 \subseteq [I] \land G.rfi ; G.po^2 \subseteq [G.W] \cup (G.rf ; [C])^2 \cup G.hb^2

\[ G.rf ; [rng(G.rfe ; [G.E^{2acq}])] \cup G.po^2 = G.rfe ; [G.E^{2acq}] \cup G.po^2 \]

\[ \square \]

**Lemma F.12.** Let \( G \vdash \langle C, I \rangle \rightarrow \langle C', I' \rangle \). Then we have:

\[ G.D_{\langle C', I' \rangle} = G.D_{\langle C, I \rangle} \cup \]
\[ \cup C' \setminus C \cup I' \setminus I \cup \]
\[ \cup dom(G.rfi^2 ; G.ppo ; [I' \setminus I]) \cup \]
\[ \cup rng([I' \setminus I] ; G.rfi) \]

**Proof.** Follows immediately from the definition of determined events and the monotonicity of the traversal step. \( \square \)

**Lemma F.13.** Let \( G \vdash \langle C, I \rangle \rightarrow_{t} \langle C', I' \rangle \). Then we have:

\[ G.vf_{\langle C', I' \rangle} = G.vf_{\langle C, I \rangle} \cup \]
\[ \cup [G.W \cup (G.rf ; [C'] \setminus C)]^2 \cup G.hb^2 \cup \]
\[ \cup G.rf ; [dom(G.ppo ; [I'] \setminus I]) \cup G.po^2 \cup \]

**Proof.** Follows from the Lemma F.11 and the monotonicity of traversal step. \( \square \)

**Lemma F.14.** For any valid traversal configuration the following is true.

\[ dom(G.rfi ; G.ppo ; [I]) \subseteq G.W \setminus rng(G.ppo) \cup rng([dom(G.ppo ; [I])] ; G.ppo) \]

**Proof.** Consider \( w_{iss} \in I \) and \( w, r \) s.t. \( \langle w, r \rangle \in G.rfi \) and \( \langle r, w_{iss} \rangle \in G.ppo \). Then consider the cases for \( w \); it either has some \( G.ppo \)-preceding reads or not. In the later case \( w \in G.W \setminus rng(G.ppo) \). In the former case consider some \( r_w \), s.t. \( \langle r_w, w \rangle \in G.ppo \). But then

\[ \langle r_w, w_{iss} \rangle \in G.ppo \cup G.rfi \cap G.ppo ; [I] \subseteq G.ppo ; [I] \]

(the last inequality follows from the definition of \( G.ppo \)).

Therefore we have that \( w \in rng([dom(G.ppo ; [I])] ; G.ppo) \). \( \square \)

**Definition F.15.** Let \( G \vdash \langle C, I \rangle \rightarrow \langle C', I' \rangle \) and \( E \doteq G.R \cap (C' \setminus C) \cup dom(G.ppo ; [I' \setminus I]) \).

Since \( ppo \subseteq po \) we have that all events from \( E \) belong to the same thread, i.e., there exists \( t \in \text{Tid} \) such that for all \( e \in E \) we have \( G.tid(e) = t \). Therefore all the events in \( E \) can be ordered according to their \( po \) order. That is \( E = \{ e_0, ..., e_n \} \), such that \( \langle e_i, e_{i+1} \rangle \in po \) for all \( 0 \leq i \leq n \).

Then we define \( G.D^l_{\langle C', I' \rangle}, G.vf^l_{\langle C', I' \rangle} \) and \( G.sf^l_{\langle C', I' \rangle} \) as follows.
• $G.D^0_{(C',P')} = G.D_{(C,I)}$
• $G.D^i_{(C',P')} = G.D^i_{(C',P')} \cup [e_i] \cup dom(G.rfi; G.ppo; [I' \setminus I]) \cap \text{rng}([e_i]; G.ppo)$ for $0 < i < n$
• $G.D^{n+1}_{(C',P')} = G.D^n_{(C',P')} \cup [e_n] \cup \text{rng}(I'[I \setminus I]; G.rfi)$
• $G.vf^0_{(C',P')} \triangleq G.vf_{(C,I)}$
• $G.vf^{i+1}_{(C',P')} \triangleq G.vf^i_{(C',P')} \cup G.rfi; [e_i]; G.poo$ for $0 < i \leq n + 1$
• $G.sf^i_{TC} \triangleq ([G.W]; (G.vf^i_{TC} \cap G.1oc) \cap (G.R) \cap (G.co; G.vf^i_{TC})$ for $0 \leq i \leq n + 1$

**Lemma F.16.** We have that:

- $G.D^{n+1}_{(C',P')} = G.D_{(C',P')}$
- $G.vf^{n+1}_{(C',P')} = G.vf_{(C',P')}$
- $G.sf^{n+1}_{(C',P')} = G.sf_{(C',P')}$

**Proof.** Follows from the Def. F.15 and Lemmas F.12 to F.14. □

**Lemma F.17.** For any $0 \leq i \leq n + 1$ $G.vf^i_{TC}; G.eco$ is irreflexive.

**Proof.** By Prop. F.5 we know that $G.vf_{(C',P')}; G.eco$ is irreflexive. But since $G.vf^i_{TC} \subseteq G.vf$ we arrive at the conclusion that $G.vf^i_{(C',P')}; G.eco$ should also be irreflexive. □

**Lemma F.18.** For $0 \leq i \leq n$ let $R_i \triangleq \text{rng}([e_i]; (po \cap =_{1oc})^2; [R])$.

Then the following equation holds.

$$G.sf^{i+1}_{TC}; [G.E \setminus R_i] = G.sf^i_{TC}; [G.E \setminus R_i]$$

**Proof.** By definition of $G.sf^{i+1}_{TC}$ we have:

$$G.sf^{i+1}_{TC}; [G.E \setminus R_i] = ([G.W]; (G.vf^{i+1}_{TC} \cap =_{1oc}) \cap [G.R]) \cap (G.co; G.vf^{i+1}_{TC}) \cap [G.E \setminus R_i] = ([G.W]; ((G.vf^{i+1}_{TC} \cap =_{1oc}) \cap [G.R \setminus R_i]) \cap (G.co; (G.vf^{i+1}_{TC} \cap =_{1oc}) \cap [G.R \setminus R_i]$$

Therefore it is sufficient to show that $(G.vf^{i+1}_{TC} \cap =_{1oc}); [G.R \setminus R_i] = (G.vf^i_{TC} \cap =_{1oc}); [G.R \setminus R_i]$. Thus by definition of $G.vf^{i+1}_{TC}$ we have:

$$G.vf^{i+1}_{TC}; [G.E \setminus R_i] = (G.vf^i_{(C',P') \cap =_{1oc}) \cap [G.E \setminus R_i] \cup G.rfi; [e_i]; (G.poo \cap =_{1oc}) \cap [G.E \setminus R_i]$$

Then it is easy to see that $G.rfi; [e_i]; (G.poo \cap =_{1oc}) \cap [G.E \setminus R_i] = \emptyset$. □

**Lemma F.19.** For any $0 \leq i \leq n$ we have the following.

$$G.sf^{i+1}_{TC}; [G.D^i_{TC}] = G.sf^i_{TC}; [G.D^i_{TC}]$$

**Proof.** By Lemma F.18 it is sufficient to show that $G.D^i_{TC} \subseteq G.E \setminus R_i$ or equivalently $G.D^i_{TC} \cap R_i = \emptyset$. Also $R_i \subseteq G.R$ and thus the later is equivalent to $G.R \cap G.D^i_{TC} \cap R_i = \emptyset$. Then by induction on $i$ we can prove $G.R \cap G.D^i_{TC} \cap R_i = G.R \cap G.D^i_{TC} \cap R_i$. Finally, it remains to use Lemma F.10 and unfold the definition of $R_i$. □

**Lemma F.20.** Let $0 \leq i \leq n$. Then we have that:

$$G.sf^{i+1}_{(C',P')}; [e_i] = G.rfi_{(C',P')}; [e_i]$$

Or in other words, stable justification for event $e_i$ coincides with the reads-from relation.
Proof. Since \(G.\text{sjf}\) is functional and complete for read events it is sufficient to prove
\[
G.\text{rf}(C,P) ; \{e_i\} \subseteq G.\text{sjf}^{i+1}(C,P) ; \{e_i\}
\]
It is easy to see that \(G.\text{rf}(C,P) ; \{e_i\} \subseteq G.\text{vf}^{i+1}(C,P) ; \{e_i\}\). Let \(\langle w, e_i \rangle \in G.\text{rf}(C,P) ; \{e_i\}\). Then it is left to show that \(\langle w, e_i \rangle \notin G.\text{co} ; G.\text{vf}^{i+1}(C,P)\). Suppose otherwise. Then there exists \(w'\) s.t. \(\langle w, w' \rangle \in G.\text{co}\) and \(\langle w', e_i \rangle \in G.\text{vf}^{i+1}(C,P)\). It cannot be the case that \(\langle w', e_i \rangle \in G.\text{rf} ; \{e_i\} ; G.\text{po}\) so therefore \(\langle w', e_i \rangle \in G.\text{vf}^{i+1}(C,P)\). But we also have \(\langle e_i, w' \rangle \in G.\text{fr} \subseteq G.\text{eco}\) which contradicts Lemma F.17. \(\square\)

Lemma F.21. For any \(0 \leq i \leq n + 1\) we have the following.
\[
G.\text{sjf}^i_C ; [G.\text{D}^i_C] = G.\text{rf} ; [G.\text{D}^i_C]
\]
Proof. Proof goes straightforwardly by induction on \(i\) using Prop. F.7 and Lemmas F.19 and F.20. \(\square\)

Lemma F.22. Let \(0 \leq i \leq n\). Then we have that:
\[
G.\text{sjf}^{i+1}(C,P) ; [\text{rng}(\{e_i\}) ; G.\text{po} \cap =_{G.1oc}] \subseteq G.\text{sjf}^i(C,P) \cup G.\text{sjf}^{i+1}(C,P) ; \{e_i\} ; G.\text{po}
\]
Proof. Consider \(\langle w, r \rangle \in G.\text{sjf}^{i+1}(C,P) \subseteq G.\text{vf}^{i+1}(C,P) = G.\text{vf}^i(C,P) \cup G.\text{rf} ; \{e_i\} ; G.\text{po}\). If \(\langle w, r \rangle \in G.\text{rf} ; \{e_i\} ; G.\text{po}\) then by Lemma F.20 \(\langle w, r \rangle \in G.\text{sjf}^{i+1}(C,P) ; \{e_i\} ; G.\text{po}\) and we are done. Otherwise \(\langle w, r \rangle \in G.\text{vf}^i(C,P)\). But then
\[
\langle w, r \rangle \in G.\text{vf}^i(C,P) \setminus G.\text{co} ; G.\text{vf}^i(C,P) \subseteq G.\text{vf}^i(C,P) \setminus G.\text{co} ; G.\text{vf}^i(C,P) = G.\text{sjf}^i(C,P)
\]
\(\square\)

Lemma F.23. For any \(0 \leq i \leq n\) the following is true.
\[
G.\text{sjf}^i(C,P) ; \{e_i\} \subseteq G.\text{sjf}^i(C,P) ; G.\text{hb}
\]
Proof. Let \(\langle w, e_i \rangle \in G.\text{sjf}^i(C,P) \subseteq G.\text{vf}^i(C,P)\). By Lemma F.11:
\[
G.\text{vf}^i(C,P) = [G.\text{w}; [G.\text{rf} ; C]]^7; G.\text{hb}^\ast \cup G.\text{rf} ; [\text{dom}(G.\text{ppo} ; \{e_0, \ldots, e_{l-1}\})]; G.\text{po} \cup G.\text{fr}; [G.\text{E}^\text{acq}]; G.\text{po}
\]
It is left to notice the following facts
- \(G.\text{rf} ; [C] = G.\text{sjf}^i(C,P) ; [C]\);
- \(G.\text{rf} ; \{e_j\} = G.\text{sjf}^i(C,P) ; \{e_j\}\) for all \(j < i\) (by Lemma F.21);
- \(G.\text{fr}; [G.\text{E}^\text{acq}] = G.\text{sfj}^i(C,P) ; [G.\text{E}^\text{acq}]\) (also by Lemma F.21). \(\square\)

Lemma F.24. Let \(0 \leq i \leq n\). Then we have that:
\[
G.\text{sjf}^{i+1}(C,P) \setminus ((G.\text{sjf}^{i+1}(C,P))^7 ; G.\text{hb}) ; [\text{rng}(\{e_i\}) ; G.\text{po} \cap =_{G.1oc}] = \]
\[
= G.\text{sjf}^i(C,P) \setminus ((G.\text{sjf}^i(C,P))^7 ; G.\text{hb}) ; [\text{rng}(\{e_i\}) ; G.\text{po} \cap =_{G.1oc}]
\]
Proof. From left to right \(\subseteq\) it follows from the Lemma F.22. In the opposite direction, consider \(w, r\) s.t.
- \(\langle w, r \rangle \in G.\text{sjf}^i(C,P)\);
- \(\langle w, r \rangle \notin (G.\text{sjf}^i(C,P))^7 ; G.\text{hb}\);
- \(r \in \text{rng}(\{e_i\}) ; G.\text{po} \cap =_{G.1oc}\).
Consider the write \(w'\) s.t. \(\langle w', r \rangle \in G.\text{sjf}^{i+1}(C,P)\). By Lemma F.22 either
- \(\langle w', r \rangle \in G.\text{sjf}^i(C,P)\), or
\[ \langle w', r \rangle \in G.\text{sjf}^{i+1}_{TC}; [e_i]; G.\text{po}. \]

In the former case we conclude that \( w' = w \) by functionality of \( G.\text{sjf}^i_{TC} \).

In the latter case we notice that \( \langle w', r \rangle \in G.\text{sjf}^{i+1}_{TC}; [e_i]; G.\text{po} \) implies that the viewfront \( G.\text{vf}^{i+1}_{TC} \) does not increase between \( e_i \) and \( r \). But that means viewfront \( G.\text{vf}^i_{TC} \) also cannot increase and thus it has to be \( \langle w, r \rangle \in G.\text{sjf}^i_{TC}; [e_i]; G.\text{po} \). From that using Lemma F.23 we can derive that \( \langle w, r \rangle \in (G.\text{sjf}^i_{TC})'; G.\text{hb} \) which contradicts our assumption.  

\[ \]  

**Lemma F.25.** For any \( 0 \leq i \leq n + 1 \) we have the following.

\[ G.\text{sjfi}^i_{TC} = G.\text{rfi} \]

**Proof.** We prove this by induction on \( i \). In the base case \( i = 0 \) we arrive at the conclusion by noticing that \( G.\text{sjf}^0_{TC} = G.\text{sjf}_{TC} \) and by using Prop. F.6. For the induction step consider \( \langle w, r \rangle \in G.\text{sjf}^{i+1}_{TC} \). Either \( r \in R_i \) or \( r \notin R_i \). In the latter case by Lemma F.18 we have that \( G.\text{sjf}^{i+1}; [r] \in G.\text{sjf}^i_{TC}; [r] \) and we arrive at the conclusion by using inductive assumption.

Otherwise consider the cases. If \( r = e_i \) then by Lemma F.20 \( G.\text{sjf}^{i+1}_{TC}; [e_i] = G.\text{rfi}; [e_i] \). If \( r \in \text{rng}([e_i]; G.\text{po} \cap =G.\text{lo}) \) then by Lemma F.22 either

- \( G.\text{sjf}^{i+1}_{TC}; [r] = G.\text{sjf}^i_{TC}; [r] = G.\text{rfi}; [r] \) (the last equality due to inductive assumption),

or

- \( G.\text{sjf}^{i+1}_{TC}; [r] \subseteq G.\text{sjf}^i_{TC}; [e_i]; G.\text{po} \), and given that \( G.\text{sjf}^{i+1}; [e_i] = G.\text{rfi}; [e_i] \) we conclude that \( G.\text{sjf}^{i+1}_{TC}; [r] = G.\text{rfi}; [r] \)

\[ \]  

**Lemma F.26.** For any \( 0 \leq i \leq n + 1 \) we have the following.

\[ G.\text{sjf}^i_{TC} = [I]; G.\text{sjfe}^i_{TC} \cup G.\text{sjf}^i_{TC} \]

**Proof.** This lemma can be proven similarly as Lemma F.25 using Prop. F.8.

\[ \]  

**Lemma F.27.** For any \( 0 \leq i \leq n \) we have the following.

\[ G.\text{sjf}^{i+1}_{TC}; [G.E \setminus G.E_i] = G.\text{sjf}^i_{TC}; [G.E \setminus G.E_i] \]

**Proof.** This lemma can be proven similarly as Lemma F.25 using Prop. F.9.

Similarly as we defined a weaker version of the simulation relation \( I_{\text{weak}}^i(P, T, G, TC, TC', S, X) \), we need to make some changes to the simulation relation for the certification branch and introduce \( I_{\text{weak}}^{i+1}\text{cert}(P, T, G, \langle C, I \rangle, \langle C', I' \rangle, S, X, Br_{i+1}) \).

- In 1 instead of \( I(P, T \setminus t, G, \langle C, I \rangle, S, X) \) we take \( I_{\text{weak}}^i(P, T \setminus t, G, \langle C, I \rangle, \langle C', I' \rangle, S, X) \)

- In 3a we say instead that the \( Br_{i+1} \) certification branch is equal to a branch of the thread \( t \) in the configuration \( X \) up to the event \( e_i \).

- In 4a instead of \( G.D_{\langle C, I' \rangle} \) we take \( G.D_{\langle C, I' \rangle}^{i+1} \).

- \( \forall e \in Br_{i+1} \cap \left[ G.D_{\langle C, I' \rangle}^{i+1} \right] . S.\text{val}(e) = G.\text{val}(\left[ e \right]) \)

- In 5a instead of \( G.\text{sjf}_{\langle C, I' \rangle} \) we take \( G.\text{sjf}_{\langle C, I' \rangle}^{i+1} \).

- \( \left[ S.\text{jf}; [Br_{i+1}] \right] \subseteq G.\text{sjf}_{\langle C, I' \rangle}^{i+1} \)

Next, in order to modify the proof of F.3 we need to show the following.

(1) Simulation relation \( I(P, T, G, TC, S, X) \) implies weak simulation relation \( I_{\text{weak}}^0(P, T, G, TC, TC', S, X) \).
(2) Given that weak simulation relation holds, it is possible to start a construction of the next certification branch, i.e., from $I_{\text{weak}}^1(P, T, G, TC, TC', S, X)$ we can prove that $I_{\text{weak}}^{i+1}(P, T, G, TC, TC', S, X, Br_{i+1})$ holds, where $Br_{i+1} = X|_t \cap \|G.p_o; [e_i]\|$

(3) It is possible to add an event to the certification branch preserving the simulation relation and \textbf{NO-BAIT-AND-SWITCH}, i.e., if $I_{\text{weak}}^i(P, T, G, TC, TC', S, X, Br_i)$ holds and $S \xrightarrow{e} S'$, s.t. $\text{dom}(S'.p_o; [e]) = Br_i$ then $I_{\text{weak}}^i(P, T, G, TC, TC', S', X, Br_i \cup \{e\})$ and $S'$ satisfies \textbf{NO-BAIT-AND-SWITCH} assuming that $S$ also satisfies it.

(4) A complete certification branch is possible to establish the weak simulation relation, i.e., from $I_{\text{weak}}^i(P, T, G, TC, TC', S, X, Br_i)$ where $\|Br_i\| = C' \cup \text{dom}(G.p_o^2; [I'])$ we can prove that $I_{\text{weak}}^i(P, T, G, TC, TC', S, X')$ holds, where $X' \neq X \setminus X|_t \cup Br_i$.

(5) Given the complete last certification branch it is possible to establish the regular simulation relation, i.e., from $I_{\text{weak}}^{i+1}(P, T, G, TC, TC', S, X, Br_{i+1})$ where $\|Br_{i+1}\| = C' \cup \text{dom}(G.p_o^2; [I'])$ we can prove that $I(P, T, G, TC', S, X')$ holds, where $X' \neq X \setminus X|_t \cup Br_{i+1}$.

In other words, in order to simulate a traversal step, we start the construction of the certification branches. We show that upon the completion of each certification branch we can restore weak version of the simulation relation and from that we can start the construction of the next branch. Finally, upon the completion of the last branch we restore the simulation relation.

The next few lemmas formalize this idea.

**Lemma F.28.** $I(P, G, TC, S, X)$ and $G \xrightarrow{T} TC' \Rightarrow \exists I_{\text{weak}}^0(P, G, TC, TC', S, X)$.

**Proof.** Holds trivially since by definition $G.D_{(C', I')}^0 = G.D_{(C, I)}^0$ and $G.sjf^0_{(C', I')} = G.sjf_{(C, I)}^0$. □

**Lemma F.29.** Suppose $I_{\text{weak}}^i(P, T, G, TC, TC', S, X)$ holds, and $G \xrightarrow{T} TC \Rightarrow TC'$.

Then $I_{\text{weak}}^i(P, T, G, TC, TC', S, X, Br_{i+1})$ holds, where $Br_{i+1} = X|_t \cap \|\text{dom}(G.p_o; [e_i])\|$.

**Proof.** We adapt the proof of a similar lemma for $I_{\text{cert}}$ from [Moiseenko et al. 2020]. We need to show that each property of $I_{\text{weak}}^i$ holds.

- 1 and 2 hold trivially by our assumptions.
- 3 holds trivially by our choice of $Br_{i+1}$:
  
  $Br_{i+1} \cap \|\text{dom}(G.p_o; [e_i])\| = X|_t \cap \|\text{dom}(G.p_o; [e_i])\|

- 3 holds since $\text{dom}(p_o; [e_i]) \subseteq C' \cup \text{dom}(G.p_o^2; [I'])$.
- To see why 4a holds first consider the following equation:
  
  $Br_{i+1} \cap \|G.D_{TC}^{i+1}\| = X|_t \cap \|\text{dom}(G.p_o; [e_i])\| \cap \|G.D_{TC}^{i+1}\|$

  and for $e \in X \cap \|G.D_{TC}^i\|$ we know that $S.\text{lab}(e) = G.\text{lab}(\|\text{e}\|)$ by the property 4a of certification simulation relation.
- 5a holds because
  
  $\|S.sjf; [Br_{i+1}]\| = \|S.sjf; [X \cap \|\text{dom}(G.p_o; [e_i])\|]\| \subseteq G.sjf_{TC}^i; [\text{dom}(G.p_o; [e_i])] \subseteq G.sjf_{TC}^{i+1}$

  The last inequality follows from lemma F.18 and the fact that
  
  $\text{dom}(G.p_o; [e_i]) \subseteq \mathcal{R}_i = G.E \setminus \text{rng}([e_i]; (G.p_o \cap =G.\text{lloc})^2; [G.R])$

- 6a holds by our choice of $Br_{i+1}$:
  
  $Br_{i+1} \cap \|I\| = X|_t \cap \|\text{dom}(G.p_o; [e_i])\| \cap \|I\| \subseteq \text{dom}(\text{S.e}^2; [X])$
6b also holds by our choice of $\mathbf{Br}_{i+1}$:
$$X \cap [I \cap [Br_{i+1}]] \cap [I \cap [X_I] \cap dom(G.p_0; [e_i])] \subseteq$$
$$\subseteq X \cap [I \cap [X_I] \cap dom(G.p_0; [e_i])] \subseteq$$
$$\subseteq X_I \cap [I \cap dom(G.p_0; [e_i])] \subseteq Br_{i+1} \cap [I] \subseteq dom(S.ew^2; [Br_{i+1}])$$

7a and 7b follow from property 7b of the simulation relation and the fact that $Br_{i+1} \subseteq X_I$. 

\[\]

**Lemma F.30.** For $1 \leq i \leq n + 1$ if $I_i^{\text{cert}}(P, T, G, TC, TC', S, X, Br_i)$ holds and $S \xrightarrow{\text{weak}} S'$, s.t.

- $dom(S'.po; [e]) = Br_i$;
- $[e] \in C' \cup dom(G.p_0^2; [I'])$;
- $[S'.jfe; [e]] \subseteq G.sjf^f_{TC'}$

then $I_i^{\text{weak}}(P, T, G, TC, TC', S', X, Br_i \cup \{e\})$ holds.

**Proof.** We need to show that each property of $I_i^{\text{cert}}$ holds.

- To show 1 we refer to the proof of the similar statement in [Moiseenko et al. 2020]. Since the only difference between $I_i(P, T \setminus t, G, \langle C, I \rangle, S, X)$ and $I_{\text{weak}}^i(P, T \setminus t, G, \langle C, I \rangle, S, X)$ is that the later uses $G.D_i^{f(C, I')}$ and $G.sjf^f_{TC'}$ instead of $G.D_i(C, I)$ and $G.sjf^f_{TC}$ the proof proceeds in a similar vein. The only significant point we want to emphasize is the proof of 5c. To show that $dom(S'.jfe) \subseteq dom(S.ew^2; [X \cap [I]])$ the original proof utilized the fact that $[S'.jfe; [e]] \subseteq G.sjf^f_{TC'}$ and property F.8. We instead note that $[S'.jfe; [e]] \subseteq G.sjf^f_{TC'}$ and use similar Lemma F.26.

- 2 follow immediately from $I_i^{\text{cert}}(P, G, TC, TC', S, X, Br_i)$.

- 3a holds because

$$(Br_i \cup \{e\}) \cap [\text{dom}(G.p_0; [e_{i-1}])] = Br_i \cap [\text{dom}(G.p_0; [e_{i-1}])] = X_I \cap [\text{dom}(G.p_0; [e_{i-1}])].$$

- 3b holds trivially by the precondition $[e] \in C' \cup dom(G.p_0^2; [I'])$.

- 4 and 5 hold by the choice of $e$, as we chose labels of events and their justification according to $G.sjf^f_{TC'}$ and by lemma F.19 and proposition F.7 we have $G.sjf^f_{TC'} = G.rf; G.D_i^{f(C, I')}$. 

- To show 6 we note that $I \subseteq G.D_i(C, I) \subseteq G.D_i^{f(C, I')}$ and thus if $[e] \in I$ then $S'.val(e) = G.val([e])$. Similarly, similarly as it is done in [Moiseenko et al. 2020], we can pick an equivalent write in $X_I$, s.t. the labels of two writes would be equal. As we have shown above using the properties of $G.D_i^{f(C, I')}$ both writes would have label equal to $G.val([e])$.

Since we update $S'.co$ order following similar rules as in [Moiseenko et al. 2020], the proof of 7 is not affected by our modifications and thus this property remains valid. 

- 8 follows immediately from the same properties of $I_i^{\text{weak}}(P, T, G, TC, TC', S, X')$ which is implied from $I_i^{\text{cert}}(P, T, G, TC, TC', S, X, Br_i)$ by 1.

**Lemma F.31.** For $1 \leq i \leq n$ if $I_i^{\text{cert}}(P, T, G, TC, TC', S, X, Br_i)$ then $I_i^{\text{weak}}(P, T, G, TC, TC', S, X', Br_i)$ holds, where $X' \triangleq X \setminus X_I \cup Br_i$.

**Proof.** We adapt the proof of a similar lemma for $I_i^{\text{cert}}$ from [Moiseenko et al. 2020]. We need to show that each property of $I_i^{\text{weak}}$ holds.

- First we note that 2a, 3, 4, 5a, 5c, 6, 7a, and 8 follow immediately from the same properties of $I_{\text{weak}}^{i-1}(P, T, G, TC, TC', S, X')$ which is implied from $I_{\text{weak}}^{i}(P, T, G, TC, TC', S, X, Br_i)$ by 1.

- 1 holds trivially due to precondition $[Br_i] = C' \cup dom(G.p_0^2; [I'])$ and our choice of $X'$. 

Proc. ACM Program. Lang., Vol. 6, No. OOPSLA2, Article 152. Publication date: October 2022.
• 2b holds because we have
\[ X' \cap \left[ G.D^{i-1}(C',J') \right] = (X \setminus X[I]) \cap G.D^{i-1}(C',J') \cup B_{r_i} \cap \left[ G.D^{i-1}(C',J') \right] \]
For left subset we derive the required property from 2b of \( I_{\text{weak}}^{i-1}(P, T, G, TC, TC', S, X') \), for the right subset we derive it from 4a of \( I_{\text{weak}}^{i}(P, T, G, TC, TC', S, X, Br_i) \).
• To see why 5b holds first note that.
\[ \left[ S.jsf ; [X \setminus X[I]] \right] = \left[ S.jsf ; [X \setminus X[I]] \right] \cup \left[ S.jsf ; [Br_i] \right] \]
Then \( \left[ S.jsf ; [Br_i] \right] \subseteq G.jsf_{TC}^{i-1} \) is implied by 5a of \( I_{\text{weak}}^{i}(P, T, G, TC, TC', S, X, Br_i) \). Next using 5b of \( I_{\text{weak}}^{i-1}(P, T, G, TC, TC', S, X') \) and Lemma F.27 we prove the following.
\[ \left[ S.jsf ; [X \setminus X[I]] \right] \subseteq G.jsf_{i-1} \]
• 7b can be proven by the similar reasoning using property 7a of \( I_{\text{weak}}^{i}(P, T, G, TC, TC', S, X, Br_i) \) and 7b of \( I_{\text{weak}}^{i-1}(P, T, G, TC, TC', S, X') \).

\[ \square \]

**Lemma F.32.** If \( I_{\text{weak}}^{n+1}(P, T, G, TC, TC', S, X, Br_{n+1}) \) holds and \( \left[ Br_{n+1} \right] = C' \cup \text{dom}(G;\text{po}^2 ; I') \) then \( I(P, T, G, TC', S, X') \) holds, where \( X' \neq X \setminus X[I] \cup Br_{n+1} \).

**Proof.** We note that due to lemma F.16 \( I_{\text{weak}}^{n+1}(P, T, G, TC, TC', S, X, Br_{n+1}) \) is equivalent to \( I_{\text{weak}}^{n}(P, T, G, TC, TC', S, X, Br_{n+1}) \) and thus we can reuse the proof of the similar lemma from [Moiseenko et al. 2020].

We are now moving to the main lemma of our development, that is we are going to show that the axiom NO-BAIT-AND-SWITCH is preserved during the construction of the event structure.

**Lemma F.33.** Suppose S satisfies NO-BAIT-AND-SWITCH. Assume \( S \xrightarrow{e} S' \) and \( S'.ew = S.ew \). Then \( S' \) also satisfies NO-BAIT-AND-SWITCH.

**Proof.** Note that \( S.E \) is prefix-closed w.r.t. \( \text{po} \) and \( \text{jsf} \), i.e., \( S.\text{po}' ; [S.E] = S.\text{po} \) and \( S.\text{jsf}' ; [S.E] = S.\text{jsf} \). Therefore \( S'.\text{jsf}' ; S'.\text{po} ; S'.ew = S.jsf ; S.\text{po} ; S.ew \). From that we can derive the following.
\[(S'.\text{jsf} \setminus (S'.\text{jsf}' ; S'.\text{hb}))) ; S'.\text{po} ; S'.ew = (S.\text{jsf} \setminus (S.\text{jsf}' ; S.\text{hb}))) ; S.\text{po} ; S.ew \subseteq S.jsf ; (S.\text{po} \cup S.\text{lbp}) \subseteq S'.\text{jsf} ; (S'.\text{po} \cup S'.\text{lbp}) \]
The last inequality follows from the fact that \( \text{po} \), \( \text{jsf} \) and \( \text{lbp} \) are monotone w.r.t. addition of new events \( S \xrightarrow{e} S' \).

**Lemma F.34.** For \( 0 \leq i \leq n \) if \( I_{\text{weak}}^{i+1}(P, T, G, TC, TC', S, X, Br_{i+1}) \) holds, \( S \xrightarrow{e} S' \), s.t.
- \( \text{dom}(S'.\text{po} ; [e]) = Br_{i+1} \);
- \( \left[ e \right] \subseteq C' \cup \text{dom}(G;\text{po}^2 ; I') \);
- \( \left[ S'.\text{jsf} ; [e] \right] \subseteq G.jsf_{TC}^{i} \)
and if \( S \) satisfies NO-BAIT-AND-SWITCH then \( S' \) also satisfies NO-BAIT-AND-SWITCH.

**Proof.** Let us note that it is sufficient to consider only the case when \( e \in W \). Otherwise by construction \( S'.ew = S.ew \) and using Lemma F.33 we can immediately conclude that \( S' \) satisfies NO-BAIT-AND-SWITCH. So let \( e \in W \).

By the construction of [Moiseenko et al. 2020, §4.3.2] the \( ew \) relation is updated during the step \( S \xrightarrow{e} S' \) when there exists some issued write event \( w \) belonging to the configuration \( X \) such that its images of \( w \) and \( e \) in the graph \( G \) are equal, i.e., \( w \in X \cap [I] \) and \( \left[ e \right] = \left[ e \right] \). In this case \( e \) is
attached to the $S.\text{ew}^*$ equivalence class of $w$ by adding a new $\text{ew}$-edge from $w$ to $e$ (see also property 6 of simulation relation for certification branch to get the idea about this construction).

Therefore, in order to prove that $S'$ satisfies NO-BAIT-AND-SWITCH we need to consider justification writes for reads in the two conflicting branches $Br_i = X[i]$ and $Br_{i+1}$. By Lemma F.30 we know that $F_{ir}^{\text{cert}}(P,T,G,TC,TC',S',X,Br_{i+1} \cup \{e\})$ holds and thus by property 5b of weak simulation relation and property 5a of simulation relation for certification branch we know that

- $\lbrack S.\text{jf} ; [Br_i]\rbrack \subseteq G.\text{sjf}^{i}_{(C',P')}$, and
- $\lbrack S.\text{jf} ; [Br_{i+1}]\rbrack \subseteq G.\text{sjf}^{i+1}_{(C',P')}$

By Lemma F.18 we also know that:

$$G.\text{sjf}^{i+1}_{(C',P')} : [G.E \setminus \mathcal{R}_i] = G.\text{sjf}^{i}_{TC} : [G.E \setminus \mathcal{R}_i]$$

where $\mathcal{R}_i \triangleq \text{rng([} e_i \cup \text{rng([} I' \setminus I) ; G.\text{rfi}]) ; (\text{po} \cap =_{\text{loc}}^2 ; [R])$. Put simply, for all read events except those in $\mathcal{R}_i$ two conflicting branches $Br_i$ and $Br_{i+1}$ pick the same justification writes and thus for these reads the conditions of NO-BAIT-AND-SWITCH are met.

Therefore it is sufficient to consider justification writes in $\mathcal{R}_i$.

Let us first consider events $\text{po}$-succeeding event $e_i$. Let $r$ be such event: $\langle e_i,r \rangle \in (G.\text{po} \cap =_{\text{loc}}^2 ; [G.R])$. Using Lemma F.24 we derive that

$$G.\text{sjf}^{i+1}_{(C',P')} \setminus \langle (G.\text{sjf}^{i+1}_{(C',P')})^2 ; G.\text{hb} \rangle ; [r] = G.\text{sjf}^{i}_{(C',P')} \setminus \langle (G.\text{sjf}^{i}_{(C',P')})^2 ; G.\text{hb} \rangle ; [r]$$

But that means justification writes for reads in $Br_{i+1}$ and $Br_i$ corresponding to $r$ should either be taken from $S'.\text{jf}$; $S'.\text{hb}$ prefix or be equal in two branches, therefore the conditions of NO-BAIT-AND-SWITCH are met for $r$.

It is left to consider the event $e_i$ itself. Let $r_i$ and $r_{i+1}$ be corresponding events in $Br_i$ and $Br_{i+1}$ respectively.

First, we consider justification write for $r_i$. By Lemma F.23 it should be the case that

$$\lbrack S'.\text{jf} ; [r_i]\rbrack = \lbrack S.\text{jf} ; [r_i]\rbrack \subseteq G.\text{sjf}^{i}_{(C',P')} ; [e_i] \subseteq [G.W] ; (G.\text{rf} ; [C])^2 ; G.\text{hb}$$

But then it should also be the case that:

$$S.\text{jf} ; [r_i] \subseteq ([X] ; S.\text{rf} ; [X \cap [C]])^2 ; S.\text{hb} = ([X] ; (S.\text{ew}^* ; S.\text{jf}) \setminus S.\text{cf} ; [X \cap [C]])^2 ; S.\text{hb}$$

From that we can conclude that

$$S'.\text{jf} ; [r_i] = S.\text{jf} ; [r_i] \subseteq S.\text{jf} ; S.\text{hb} \subseteq S'.\text{jf} ; S'.\text{hb}$$

Therefore $r_i$ is justified locally and thus NO-BAIT-AND-SWITCH is satisfied for $r_i$.

Finally, we need to show that $r_{i+1}$ and $w$ form load buffering pattern, i.e., $\langle r_{i+1},w \rangle \in \text{lbpat}$ (recall that $w$ is an equivalent write we picked for $e$, i.e., $\langle w,e \rangle \in S'.\text{ew}$). If we do this then clearly conditions of NO-BAIT-AND-SWITCH are met for $r_{i+1}$.

Recall the definition of the load buffering pattern

$$\text{lbpat} \triangleq \text{cf}_{\text{imm}} \setminus [\text{rng}(jf \cap (jf^2 ; \text{hb})) ; \text{po}]$$

By property 3a of simulation relation for certification branch we know that $Br_i$ and $Br_{i+1}$ are equal up to event $e_i$ and thus $r_i$ and $r_{i+1}$ should be in immediate conflict: $\langle r_{i+1},r_i \rangle \in S'.\text{cf}_{\text{imm}}$. Moreover, as we have shown $r_i$ is locally justified, i.e., $r_i \in \text{rng}(S'.\text{jf} \cap (S'.\text{jf}^2 ; S'.\text{hb}))$. Therefore $\langle r_{i+1},w \rangle \in S'.\text{cf}_{\text{imm}} \setminus [\text{rng}(jf \cap (jf^2 ; \text{hb})) ; S'.\text{po}$. 

$\Box$
G  SOUNDNESS OF PROGRAM TRANSFORMATIONS FOR WEAKESTMO2

In this section we present the proofs of the soundness of program transformations in Weakestmo2. Our development is based on the proofs for the original Weakestmo model [Chakraborty et al. 2019].

G.1  Reordering of Independent Instructions

We start with the soundness of reorderings. We consider four kind of reorderings: store/load, store/store, load/load, and load/store.

\[
\begin{align*}
\begin{alignat}{2}
& x := r_1 \hspace{1em} ; \hspace{1em} r_2 := y \quad \mapsto \quad r_2 := y \hspace{1em} ; \hspace{1em} x := r_1 & \quad & \text{store/load} \\
& x := r_1 \hspace{1em} ; \hspace{1em} y := r_2 \quad \mapsto \quad y := r_2 \hspace{1em} ; \hspace{1em} x := r_1 & \quad & \text{store/store} \\
& r_1 := x \hspace{1em} ; \hspace{1em} r_2 := y \quad \mapsto \quad r_2 := y \hspace{1em} ; \hspace{1em} r_1 := x & \quad & \text{load/load} \\
& r_1 := x \hspace{1em} ; \hspace{1em} y := r_2 \quad \mapsto \quad y := r_2 \hspace{1em} ; \hspace{1em} r_1 := x & \quad & \text{load/store}
\end{alignat}
\]

We assume that all accesses are relaxed. Handling other kinds of accesses would significantly complicate the proofs with a lot of technical details that are orthogonal to the main idea behind our reasoning.

**Theorem G.1.** Reordering transformation \( P_{\text{src}} \xrightarrow{\text{reord}(r,a,b)} P_{\text{tgt}} \) where instructions \( a \) and \( b \) form a store/load, store/store, load/load, or load/store pair of independent accesses is sound.

G.1.1  Recap of the Proof Structure. Let us first remind the proof structure for the original version of Weakestmo [Chakraborty et al. 2019, §F].

Given a Weakestmo2 consistent execution graph \( G_{\text{tgt}} \) of program \( P_{\text{tgt}} \) one need to construct a Weakestmo2 consistent execution graph \( G_{\text{src}} \) of \( P_{\text{src}} \) such that \( B(G_{\text{src}}) = B(G_{\text{tgt}}) \). To do so Chakraborty et al. [2019, §F] consider target event structure \( S_{\text{tgt}} \), s.t. \( S_{\text{tgt}} \triangleright G_{\text{tgt}} \), and build source event structure \( S_{\text{src}} \) and execution graph \( G_{\text{src}} \), s.t. \( S_{\text{src}} \triangleleft G_{\text{tgt}} \), following the construction of \( S_{\text{tgt}} \) by the operational semantics \( S_{\text{init}}(P_{\text{tgt}}) \rightarrow^* S_{\text{tgt}} \).

Similarly to the proof of the correctness of compilation mappings §F.1, the proof of the soundness of reorderings relies on the simulation argument. Chakraborty et al. [2019, §F] define the simulation relation \( I(P_{\text{src}}, P_{\text{tgt}}, G_{\text{tgt}}, S_{\text{src}}, S_{\text{tgt}}, X_{\text{src}}, \mu) \) between the source and target programs \( P_{\text{src}}, P_{\text{tgt}} \); target execution graphs \( G_{\text{tgt}} \); source and target Weakestmo consistent event structures \( S_{\text{src}}, S_{\text{tgt}} \); justified configuration of the source event structure \( X_{\text{src}} \), and function \( \mu : S_{\text{src}}.E \rightarrow S_{\text{tgt}}.E \) that maps events of the source event structure to the events of the target event structure.

The three simulation lemmas are presented below ensure that: (i) initial source and target event structures are bound by the simulation relation; (ii) each construction step of the target event structure can be simulated by the construction step of the source event structure; (iii) at the end of the simulation process, selected configuration of source event structures forms an execution graph with the same behavior as the given target execution graph.

**Lemma G.2 (Simulation $\text{Start}$).** Let \( P_{\text{src}} \) and \( P_{\text{tgt}} \) be source and target programs, s.t. \( P_{\text{src}} \xrightarrow{\text{reord}(r,a,b)} P_{\text{tgt}} \), and let \( G_{\text{tgt}} \) be Weakestmo consistent execution graph, corresponding to the target program \( P_{\text{tgt}} \). Then \( I(P_{\text{src}}, P_{\text{tgt}}, G_{\text{tgt}}, S_{\text{init}}(P_{\text{src}}), S_{\text{init}}(P_{\text{tgt}}), \{e_0^s\}, \mu_0) \) holds where

- \( S_{\text{init}}(P_{\text{src}}) \) is initial source event structure containing only the initialization event \( \{e_0^s\} \);

\(^8\)In order to simplify the proof, we have slightly modified the simulation relation presented in Chakraborty et al. [2019, §F]. In particular, instead of the relation \( M \subseteq S_{\text{src}}.E \times S_{\text{tgt}}.E \) that connects events of the source and target event structure, we use function \( \mu : S_{\text{src}}.E \rightarrow S_{\text{tgt}}.E \). We then use notations \( \lceil \cdot \rceil \) and \( \lfloor \cdot \rfloor \), similarly as in §F.2, to lift function \( \mu \) to subsets of events and component relations (such as \( p_0, j_{\text{f}}, c_{\text{oc}} \)) of the two event structures.
• \( S_{\text{init}}(P_{\text{tgt}}) \) is initial target event structure containing only the initialization event \( \{e_0^t\} \);
• \( \mu_0 \) is the initial event mapping, s.t. \( \mu_0(e^0) = e_0^t \) and \( \mu_0(e) = \perp \) for all \( e \neq e^0_0 \).

**Lemma G.3 (Weak Simulation Step).** If \( I(P_{\text{src}}, P_{\text{tgt}}, G_{\text{tgt}}, S_{\text{src}}, S_{\text{tgt}}, X_{\text{src}}, \mu) \) and \( S_{\text{tgt}} \rightarrow^* S'_{\text{tgt}} \) hold then there exist \( S'_{\text{src}}, X'_{\text{src}} \), and \( \mu' \) such that \( I(P_{\text{src}}, P_{\text{tgt}}, G_{\text{tgt}}, S'_{\text{src}}, S'_{\text{tgt}}, X'_{\text{src}}, \mu') \) and \( S_{\text{src}} \rightarrow^* S'_{\text{src}} \) hold.

**Lemma G.4 (Simulation End).** If \( I(P_{\text{src}}, P_{\text{tgt}}, G_{\text{tgt}}, S_{\text{src}}, S_{\text{tgt}}, X_{\text{src}}, \mu) \) and \( S_{\text{tgt}} \not\supset G_{\text{tgt}} \) hold, then the execution graph associated with \( X_{\text{src}} \), denoted as \( G_{\text{src}} \), has the same behavior as \( G_{\text{tgt}} \), i.e., \( B(G_{\text{src}}) = B(G_{\text{tgt}}) \).

Similarly to the case of Theorem 3.14, the main objective of our modification of the proof is to show that the simulation of the construction step (i.e., Lemma G.3) preserves the new axiom 

**NO-BAIT-AND-SWITCH.**

**G.1.2 Simulation Relation.** We next describe the simulation relation \( I \).

The simulation relation \( I \) distinguishes three subsets of events of the target event structure \( S_{\text{tgt}} \).

Giving that \( P_{\text{src}} \xrightarrow{\text{reord}(r,a,b)} P_{\text{tgt}} \) let

- \( A \subseteq S_{\text{tgt}} \cdot E \) be a subset of events obtained as a result of executing instruction \( a \);
- \( B \subseteq S_{\text{tgt}} \cdot E \) be a subset of events obtained as a result of executing instruction \( b \);
- \( C \triangleq S_{\text{tgt}} \cdot E \setminus (A \cup B) \) contain the remaining events.

The simulation \( I \) relation establishes the following properties

1. For each non-reordered event in \( S_{\text{tgt}} \) there exists a corresponding event in \( S_{\text{src}} \):
   (a) \( \forall e \in S_{\text{tgt}} \cdot E \exists A \in S_{\text{src}} \cdot E \) s.t. \( S_{\text{src}} \cdot \text{lab}(e) = S_{\text{tgt}} \cdot \text{lab}(\overline{\text{lab}(e)}) \)

2. Labels of events in \( S_{\text{src}} \) match labels of events in \( S_{\text{tgt}} \):
   (a) \( \forall e \in S_{\text{src}} \cdot E \exists A \in S_{\text{tgt}} \cdot E \) s.t. \( S_{\text{src}} \cdot \text{lab}(e) = S_{\text{tgt}} \cdot \text{lab}(\overline{\text{lab}(e)}) \)

3. Program order in \( S_{\text{tgt}} \) between events from \( A \) and \( C \) corresponds to program order in \( S_{\text{src}} \):
   (a) \( \forall e \in S_{\text{src}} \cdot E \exists A \in S_{\text{tgt}} \cdot E \) s.t. \( S_{\text{src}} \cdot \text{po}(e) = S_{\text{tgt}} \cdot \text{po}(\overline{\text{po}(e)}) \)

4. Identity relation in \( S_{\text{tgt}} \) corresponds to identity or conflict relation in \( S_{\text{src}} \):
   - \( \forall \overline{id} \in \overline{\text{src}} \cdot \text{cf} \)

5. Justified-from relation in \( S_{\text{tgt}} \) from an event belonging to \( A \) or \( C \) to an event belonging to \( C \)
   corresponds to justified-from relation in \( S_{\text{src}} \):
   (a) \( \forall e \in S_{\text{src}} \cdot E \exists A \in S_{\text{tgt}} \cdot E \) s.t. \( S_{\text{src}} \cdot \text{jf}(e) = S_{\text{tgt}} \cdot \text{jf}(\overline{\text{jf}(e)}) \)

Each justification of an event from \( A \) in \( S_{\text{src}} \) is either local (i.e., from \( jf \); \( hb \) preceding write) or the same as in \( S_{\text{tgt}} \):
(d) \( S_{\text{src}} \cdot \text{jf} ; \{ [A] \} \subseteq S_{\text{src}} \cdot \text{jf}^2 \cdot S_{\text{src}} \cdot \text{hb} \cup S_{\text{tgt}} \cdot \text{jf} \)

(6) Each pair of equivalent writes from in \( S_{\text{src}} \) between events from \( B \) corresponds to the same event or a pair of equivalent writes in \( S_{\text{tgt}} \):

(a) \([B] ; \{ S_{\text{src}} \cdot \text{ew} \} \cup [B] \subseteq S_{\text{tgt}} \cdot \text{ew}'\)

Equivalent-writes relation in \( S_{\text{tgt}} \) between write events from \( C \) corresponds to Equivalent-writes relation in \( S_{\text{src}} \):

(b) \([C] ; S_{\text{tgt}} \cdot \text{ew} ; [C] = [C] ; \{ S_{\text{src}} \cdot \text{ew} \} ; [C] \)

(7) Coherence relation in \( S_{\text{tgt}} \) corresponds to the coherence relation in \( S_{\text{src}} \):

(a) \( S_{\text{tgt}} \cdot \text{co} = \{ S_{\text{src}} \cdot \text{co} \} \)

(8) Extracted execution graph \( G_{\text{src}} \), corresponding to configuration \( X_{\text{src}} \), is equal to execution graph \( G_{\text{tgt}} \) modulo reordering of events \( a \in A \cap G_{\text{tgt}} \cdot E \) and \( b \in B \cap G_{\text{tgt}} \cdot E \)

(a) \( G_{\text{src}} \cdot E = G_{\text{tgt}} \cdot E \cap S_{\text{tgt}} \cdot E \)

(b) \( \forall e \in G_{\text{src}} \cdot E. G_{\text{src}} \cdot \text{lab}(e) = G_{\text{tgt}} \cdot \text{lab}(e) \)

(c) \( [G_{\text{src}} \cdot \text{po}] = G_{\text{tgt}} \cdot \text{po} \cdot S_{\text{tgt}} \cdot E \}

(d) \( [G_{\text{src}} \cdot \text{rf}] = G_{\text{tgt}} \cdot \text{rf} \cdot S_{\text{tgt}} \cdot E \)

(e) \( [G_{\text{src}} \cdot \text{co}] = G_{\text{tgt}} \cdot \text{co} \cdot S_{\text{tgt}} \cdot E \)

G.1.3 Adapting the proof. We need to show that simulation step respects NO-BAIT-AND-SWITCH axiom.

\[
(\text{jf} \setminus (\text{jf}^2 ; \text{hb})) ; \text{po} ; \text{ew} \subseteq \text{jf} ; (\text{po} \cup \text{lbpat})
\]

That is, we need to show that if the simulation relation \( I (P_{\text{src}}, P_{\text{tgt}}, G_{\text{tgt}}, S_{\text{src}}, S_{\text{tgt}}, X_{\text{src}}, \mu) \) holds, the source event structure \( S_{\text{src}} \) satisfies NO-BAIT-AND-SWITCH, and the construction of the target event structure makes a step \( S_{\text{tgt}} \overset{e}{\rightarrow} S'_{\text{tgt}} \), s.t. \( S_{\text{tgt}} \) and \( S'_{\text{tgt}} \) both satisfy NO-BAIT-AND-SWITCH, then there exist \( S'_{\text{src}} \), \( X'_{\text{src}} \), and \( \mu' \) such that \( S_{\text{src}} \overset{\mu}{\rightarrow} S'_{\text{src}} \), \( I (P_{\text{src}}, P_{\text{tgt}}, G_{\text{tgt}}, S'_{\text{src}}, S'_{\text{tgt}}, X'_{\text{src}}, \mu') \) holds, and \( S'_{\text{src}} \) also satisfies NO-BAIT-AND-SWITCH.

In order to prove this we consider each kind of reordering separately.

Store/Load: Consider the cases.

\[
\begin{align*}
\text{a} & : W(x, \alpha) \quad \text{b}_t : R(y, \beta) \\
\text{b}_t' & : R(y, \gamma) \quad \text{a}_t : W(x, \alpha) \\
\text{b}_s : R(y, \beta) & \quad \text{a}_t : W(x, \alpha)
\end{align*}
\]

(a) Source event structure

(b) Target event structure

Fig. 8. A fragment of the event structure construction that justifies store/load reordering. Semi-transparent events may or may not be presented in the event structures when event \( b'_2 \) is added.

- \( e_t \in B \)

We first consider the most challenging case. Let us consult Fig. 8. The target event structure adds event \( e_t = b_t : R(y, \beta) \). To simulate this step, source event structure should be first updated with an event \( a_t : W(x, \alpha) \) (if there exists \( a'_t \), s.t. \( \langle a'_t, a_t \rangle \in S_{\text{tgt}} \cdot \text{ew} \), then \( S_{\text{src}} \) should already contain \( a_s \), s.t. \( \mu(a_s) = a'_t \)), and then with an event \( b_s : R(y, \beta) \):

\[
S_{\text{src}} \overset{a_s}{\rightarrow} S'_{\text{src}} \overset{b_s}{\rightarrow} S'_{\text{src}}
\]

Note that it is indeed possible to perform the first step, because instructions \( a \) and \( b \) are independent (thus the written value \( \alpha \) of the event \( a_t : W(x, \alpha) \) cannot depend on the value of the previous event \( b_t : R(y, \beta) \)), and because it is always possible to add a write event to
the event structure (because write events do not need to be justified via \( jf \) relation). Finally, note that if event \( b_t \) depends on some promises in \( S'_{tgt} \), then in \( S'_{src} \) it should depend on the same or smaller set of promises. Indeed, suppose that in \( S'_{tgt} \) event \( b_t \) depends via \( j_o \) path on event \( a'_t \), s.t. \( \langle a'_t, a_t \rangle \in S'_{tgt}.ew \). In \( S'_{src} \) this dependency will be lost, since \( b_s \) will depend on \( a_s \). All other promises will remain the same. Similarly, write event \( b_s \) and all subsequent write events will depend on the same set of external writes (since \( b_s \) and \( b_t \) have the same justification writes). Therefore, if \( b_s \) depends on some promises it will be possible to certify them by the same certification writes as in target event structure \( S'_{tgt} \) without violation of NO-BAIT-AND-SWITCH.

- \( e_t \in A \)

Consult Fig. 8 again, and let \( e_t = a_t \). In this case we can conclude that \( S_{src} \) should already have a matching event \( a_s \). Thus \( S'_{src} = S_{src} \).

- \( e_t \in C \)

In this case construction of the source event structure can simulate the step by adding event \( e_s \) with the same label \( S_{src} \rightarrow S'_{src} \). Since \( S'_{tgt} \) satisfies NO-BAIT-AND-SWITCH then so \( S'_{src} \), because if \( e \) belongs to some certification branch in \( S'_{tgt} \) then in \( S'_{src} \) this branch should depend on the smaller or equal set of promises (as was explained above) and on the same external writes.

**Store/Store:** Consider the cases.

\[
\begin{align*}
\text{(a) Source event structure} & \quad \text{(b) Target event structure} \\
\begin{array}{c}
a_s : W(x, \alpha) \\
b_s : W(y, \beta)
\end{array} & \quad \begin{array}{c}
b_t : W(y, \beta) \\
a_t : W(x, \alpha)
\end{array}
\end{align*}
\]

Fig. 9. A fragment of event structure construction that justifies store/store reordering.

- \( e_t \in B \)

Let us consult Fig. 9. The target event structure adds event \( e_t = b_t : W(y, \beta) \). To simulate this step, events \( a_s : W(x, \alpha) \) and \( b_s : W(y, \beta) \) are added to the source event structure:

\[
S_{src} \xrightarrow{a_s} S'_{src} \xrightarrow{b_s} S'_{src}
\]

The events \( a_s \) and \( b_s \) in \( S'_{src} \) depend on the same set of promises and external writes as \( b_t \) in \( S'_{tgt} \). Therefore, each of these promise will be certified by the same certification writes as in target event structure \( S'_{tgt} \) without violation of NO-BAIT-AND-SWITCH.

- \( e_t \in A \)

Consult Fig. 8 again, and let \( e_t = a_t \). In this case we can conclude that \( S_{src} \) should already have a matching event \( a_s \). Thus \( S'_{src} = S_{src} \).

- \( e_t \in C \)

In this case construction of the source event structure can simulate the step by adding event \( e_s \) with the same label \( S_{src} \rightarrow S'_{src} \). Since \( S'_{tgt} \) satisfies NO-BAIT-AND-SWITCH then so \( S'_{src} \).

**Load/Load:** Consider the cases.

- \( e_t \in B \)

Let \( e_t = b_t : R(y, \beta) \) and let \( a_t :: R(x, \alpha) \) be the subsequent event. To simulate this step, events
\( a_s : R(x, \alpha) \) and \( b_s : R(y, \beta) \) are added to the source event structure:

\[
S_{src} \xrightarrow{} S'_{src} \xrightarrow{} S''_{src}
\]

We need to consider the cases with respect to promises which \( b_t \) and \( a_t \) depend on.

- Suppose that \( a_t \), \( b_t \) and their \( po \) predecessor up to the read event \( c_t \) depend on the same set of promises, (including the case of empty set, i.e., when these events do not depend on any promises at all). This case is depicted on Fig. 10. In this case we can conclude that all promises will be certified by the same certification writes as in target event structure \( S'_{tgt} \) without violation of \textsc{no-bait-and-switch}. Indeed, each of these promises will depend on the same external writes, because \( a_s \) and \( b_s \) have the same justification writes as \( a_t \) and \( b_t \); moreover all other reads in the certification branch remain unchanged.

\[
\begin{align*}
    c_s : R(z, y) & \xrightarrow{} c'_s : R(z, y') \\
    a_s : R(x, \alpha) & \downarrow \quad \quad \downarrow \\
    b_s : R(y, \beta) & \quad \quad \downarrow \quad \quad \downarrow \\
    \quad \quad \downarrow \quad \quad \downarrow \\
    \quad \quad p'_s : W(u, \delta)
\end{align*}
\]

(a) Source event structure

\[
\begin{align*}
    c_t : R(z, y) & \xrightarrow{} c'_t : R(z, y') \\
    b_t : R(y, \beta) & \downarrow \quad \quad \downarrow \\
    a_t : R(x, \alpha) & \quad \quad \downarrow \quad \quad \downarrow \\
    \quad \quad \downarrow \quad \quad \downarrow \\
    \quad \quad p'_t : W(u, \delta)
\end{align*}
\]

(b) Target event structure

Fig. 10. A fragment of event structure construction that justifies load/load reordering. Case when both reads \( b_t \), \( a_t \) and their \( po \) predecessors up to \( c_t \) depend on the same set of promises.

- Suppose that \( a_t \), \( b_t \) depend on the same set of promises, but their \( po \) predecessor do not depend on these promises. Also suppose that \( p'_s \) is \( po \) maximal promise. This case is depicted on Fig. 11. First, let us prove that in this case \( \alpha = \alpha' \) and thus \( a_t \) and \( a'_t \) should have the same label. Indeed, since \( a_t \) and \( a'_t \) belong to neighboring branches due to \textsc{no-bait-and-switch} it has to be that either these reads are local, i.e., they read from \( jf^2 ; hb \) prior \( co \)-maximal write, or they are external and thus should read from the same write. Therefore, it has to be the case that \( S_{src} \) already contains \( a_s \), s.t. \( \mu(a_s) = a'_t \), and thus \( S''_{src} = S_{src} \). Note that \( b_s \) in \( S'_{src} \) depends on the same set of promises as \( b_t \) in \( S'_{tgt} \). All these promises in \( S''_{src} \) will depend on the same or lesser set of external writes compared to \( S'_{tgt} \) (because the justification write of read event \( a_s \) will not be considered as external write with respect to certification branch containing \( b_s \)). Therefore these promises will be certified by the same certification writes as in the target event structure \( S'_{tgt} \) without violation of \textsc{no-bait-and-switch}.

- Suppose that \( a_t \) depends on some set of promises, but \( b_t \) does not depend on these promises. Also suppose that \( p'_s \) is \( po \) maximal promise and \( a'_t \) is read event in immediate conflict with \( a_t \). This case is depicted on Fig. 12. Event \( a_s \) in \( S'_{src} \) depends on the same set of promises as \( a_t \) in \( S'_{tgt} \). These promises will be certified by the same certification writes as in the target event structure. However, if the read event \( b_s \) is justified from external write, then in the source event structure these promises will also depend on this external write. Nevertheless, note that event \( b'_s \) in the branch that have issued the promises should read from the same write as \( b_s \), since \( \mu(b'_s) = \mu(b_s) = b_t \). Therefore, in the source event structure the promise...
Fig. 11. A fragment of event structure construction that justifies load/load reordering. Case when both reads $b_t, a_t$ depend on the same set of promises but their $po$ predecessors do not depend on these promises.

$p'_s$ and its certification write will depend on the same set of external writes, thus satisfying \textsc{no-bait-and-switch}.

Fig. 12. A fragment of event structure construction that justifies load/load reordering. Case when read $a_t$ depends on some set of promises, but $b_t$ does not depend on these promises.

- $e_t \in A$
  Consult Figures 10 to 12 again, and let $e_t = a_t$. In this case we can conclude that $S'_{src} = S_{src}$.
- $e_t \in C$
  In this case construction of the source event structure can simulate the step by adding event $e_s$ with the same label $S_{src} \xrightarrow{e_s} S'_{src}$. Since $S'_{tgt}$ satisfies \textsc{no-bait-and-switch} then so $S'_{src}$.

\textbf{Load/Store: Consider the cases.}

- $e_t \in B$
  Let $e_t = b_t : W(y, \beta)$ and let $a_t : R(x, \alpha)$ be the subsequent event. Then we need to consider the following cases.
  - Suppose that $a_t$ does not depend on $b_t$ via $jo$ path. This case is depicted on Fig. 13. Then it is possible to update source event structure by adding events $a_s : R(x, \alpha)$ and $b_s : W(y, \beta)$:
    $S_{src} \xrightarrow{a_s} S'_{src} \xrightarrow{b_s} S''_{src}$
  Note that event $a_s$ is justified by the same write in $S'_{src}$ as $a_t$ in $S'_{tgt}$. Therefore, if $a_s$ (and, possibly, $b_s$) depend on some set of promises, it will be possible to certify them by the same writes as in $S'_{tgt}$. Moreover, the set of external writes of these promises remain the same. Therefore, \textsc{no-bait-and-switch} holds.
- Suppose that \( a_t \) depend on \( b_t \) via \( Jo \) path and that \( b_t \), \( a_t \) do not depend on promises. This case is depicted on Fig. 14. In this case the source event structure is updated by addition of events \( a'_t : R(x, \gamma) \) and \( b'_t : W(y, \beta) \):

\[
S_{src} \xrightarrow{a'_t} S'_{src} \xrightarrow{b'_t} S_{src}
\]

As a justification write for \( a'_t \) the construction chooses event \( c_s : W(x, \gamma) \), that is co maximal non-conflicting write from \( jf \); \( hb \) prefix of \( a'_t \). Since \( a'_t \) and \( b'_t \) do not depend on promises, the condition NO-BAIT-AND-SWITCH is satisfied trivially.

\[
c_s : W(x, \gamma) \xrightarrow{jf} a'_t : R(x, \gamma) \xrightarrow{jf} a_s : R(x, \alpha) \\
\quad \quad b'_t : W(y, \beta) \xrightarrow{jf} b_s : W(y, \beta) \xrightarrow{jo} w'_t : W(x, \alpha)
\]

(a) Source event structure

(b) Target event structure

Fig. 14. A fragment of event structure construction that justifies load/store reordering. Case when \( b_t \) and \( a_t \) do not depend on promises.

- Suppose that \( a_t \) depend on \( b_t \) via \( Jo \) path, and that \( a_t \) depends on some set of promises, but \( b_t \) does not depend on these promises. This case is depicted on Fig. 15. In this case, due to NO-BAIT-AND-SWITCH, event \( a_t \) should be in immediate conflict with event \( a'_t \) which is justified locally, i.e., from \( jf \); \( hb \) preceding write \( c_t : W(x, \gamma) \). We can conclude that \( S_{src} \) should contain events \( a'_t : R(x, \gamma) \) and \( b'_t : W(y, \beta) \), s.t. \( \mu(a'_t) = a'_t \) and \( \mu(b'_t) = b_t \). Therefore \( S'_{src} = S_{src} \).

\[
c_s : W(x, \gamma) \xrightarrow{jf} a'_t : R(x, \gamma) \xrightarrow{jf} a_s : R(x, \alpha) \\
\quad \quad b_t : W(y, \beta) \xrightarrow{jf} b_t : W(y, \beta) \xrightarrow{jo} w_s : W(x, \alpha) \\
\quad \quad p'_t : W(u, \delta) \xrightarrow{jo} p'_t : W(u, \delta)
\]

(a) Source event structure

(b) Target event structure

Fig. 15. A fragment of event structure construction that justifies load/store reordering. Case when \( a_t \) depends on promises but \( b_t \) does not.
Suppose that $a_t$ depend on $b_t$ via $\text{jo}$ path, and also that $a_t, b_t$ depend on the same set of promises. This case is depicted on Fig. 16. Consider write $w_t$ that justifies $a_t$. Note that $w_t$ is external write, i.e., $\langle w_t, a_t \rangle \notin jf^2; \text{hb}$, or otherwise we would get $\text{po} \cup jf$ cycle $b_t \xrightarrow{\text{jo}} w_t \xrightarrow{\text{jo}} b_t$. Also note that it cannot be the case that $\langle w_t, p_t'' \rangle \in jf^2; \text{hb}$, as required by $\text{NO-BAIT-AND-SWITCH}$, because it also would imply $\text{po} \cup jf$ cycle $w_t \xrightarrow{\text{jo}} p_t'' \xrightarrow{\text{jo}} w_t$. Therefore, there should exists another certification branch neighboring to the branch containing $a_t$. In other words, there should exists event $a_t'$ in immediate conflict with $a_t$, and all subsequent events up to the event $p_t'$ certifying promise $p_t''$. Thus, the source event structure is updated by addition of a branch $a_t' \xrightarrow{\text{jo}} b_t' \xrightarrow{\text{jo}} \ldots \xrightarrow{\text{jo}} p_t'$:

$$S_{src} \rightarrow^* S'_{src}$$

Since this new branch in $S'_{src}$ mimics the corresponding branch in $S'_{tgt}$, it also satisfies $\text{NO-BAIT-AND-SWITCH}$.

Fig. 16. A fragment of event structure construction that justifies load/store reordering. Case when $a_t$ and $b_t$ depends on the same set of promises.

- $e_t \in A$
  Let $e_t = a_t : \text{R}(x, \alpha)$. We need to consider the same cases as above.
- Consult Fig. 13 again. It can be seen that $S'_{src}$ should already contain suitable event $a_s$. Therefore $S'_{src} = S_{src}$.
- Consult Fig. 14 again. In this case the source event structure is updated by addition of events $a_s : R(x, \alpha)$ and $b_s : W(y, \alpha)$:

$$S_{src} \xrightarrow{a_s} S'_{src} \xrightarrow{b_s} S'_{src}$$

These events form a new certification branch. This branch satisfies NO-BAIT-AND-SWITCH, because it only differs from the neighboring branch at the point of immediate conflict, and moreover $a'_s$ is justified locally, i.e., from $jf?: hb$ prior write.
- Consult Fig. 15 again. In this case the source event structure is updated by addition of events $a_s : R(x, \alpha)$ and $b_s : W(y, \alpha)$:

$$S_{src} \xrightarrow{a_s} S'_{src} \xrightarrow{b_s} S'_{src}$$

These events form a new certification branch mimicking the branch starting at $a_t$ in $S'_{tgt}$. All promises on which this branch depends will be certified by the same certification writes as in $S'_{tgt}$. Additionally, in $S'_{src}$ this branch also depends on the promise $b'_s$. This promise is certified by $b_t$. Therefore, the new branch satisfies NO-BAIT-AND-SWITCH.
- Consult Fig. 16 again. In this case the source event structure is updated by addition of events $a_s : R(x, \alpha)$ and $b_s : W(y, \alpha)$:

$$S_{src} \xrightarrow{a_s} S'_{src} \xrightarrow{b_s} S'_{src}$$

Similarly to the previous case, these events form a new certification branch mimicking the branch starting at $a_t$ in $S'_{tgt}$. By the same reasoning, the new branch satisfies NO-BAIT-AND-SWITCH.

$• e_t \in C$

In this case construction of the source event structure can simulate the step by adding event $e_s$ with the same label $S_{src} \xrightarrow{e_s} S'_{src}$. Since $S'_{tgt}$ satisfies NO-BAIT-AND-SWITCH then so $S'_{src}$.

### G.2 Elimination of Redundant Accesses

Next we prove soundness of eliminations.

Suppose that thread $t$ of a program $P$ contains two consecutive instructions $a ; b$ such that these instructions access the same location and one of the instructions is redundant. For example, $a$ and $b$ can be a pair of stores to the same location, then the first store is redundant because its value is overwritten by the second one. Then elimination transformation $\text{elim}(t, a, b)$ removes or replaces redundant instruction, and leaves other threads unchanged.

We consider three kind of eliminations: store/load, store/store, load/load$^9$:

- $x := r_1 ; r_2 := x$ $\leadsto$ $x := r_1 ; r_2 := r_1$ store/load
- $x := r_1 ; x := r_2$ $\leadsto$ $x := r_2$ store/store
- $r_1 := x ; r_2 := x$ $\leadsto$ $r_1 := x ; r_2 := r_1$ load/load

Again, we assume that all accesses are relaxed.

**Theorem G.5.** Elimination transformation $P_{src} \xrightarrow{\text{elim}(f, a, b)} P_{tgt}$ where instructions $a$ and $b$ form a store/load, store/store, or load/load pair of redundant accesses is sound.

Note that load/store elimination is unsound for atomic accesses both in the original Weakestmo and in C11 models [Chakraborty et al. 2019; Vafeiadis et al. 2015] and thus we do not consider it.
G.2.1 Recap of the Proof Structure. Proof of the soundness of eliminations mimics proof of the soundness of transformations [Chakraborty et al. 2019, §G]. Similarly as in §G.1.1 the construction of the source event structure simulates constructions of the target event structure. The main objective of our modification of the proof is to show that the simulation of the construction step preserves the new axiom NO-BAIT-AND-SWITCH.

G.2.2 Simulation Relation. We next describe the simulation relation $I$ used in the proof of the soundness of elimination transformations. This simulation relation has a similar form compared to the one presented in §G.2.2.

That is, the simulation relation $I (P_{src}, P_{tgt}, G_{tgt}, S_{src}, S_{tgt}, X_{src}, \mu)$ establish a connection between the source and target programs $P_{src}, P_{tgt}$; target execution graphs $G_{tgt}$; source and target Weakestmo consistent event structures $S_{src}, S_{tgt}$; justified configuration of the source event structure $X_{src}$, and function $\mu : S_{src} \rightarrow S_{tgt} \cup E$ that maps events of the source event structure to the events of the target event structure.

The simulation relation $I$ distinguishes three subsets of events of the target event structure $S_{tgt}$.

Given that $P_{src} \xrightarrow{\text{elim}(f,a)} P_{tgt}$ let

- $A \subseteq S_{tgt}.E$ be a subset of events obtained as a result of executing instruction $a$;
- $B \subseteq S_{tgt}.E$ be a subset of events obtained as a result of executing instruction $b$;
- $C = S_{tgt}.E \setminus (A \cup B)$ contain the remaining events.

Note that depending on the particular kind of elimination either $A$ or $B$ is empty, because the target program contains only one of the redundant instructions:

- in case of store/store elimination $A$ is empty;
- in case of store/load and load/load eliminations $B$ is empty.

The simulation relation $I$ establishes the following properties

(1) For each event in $S_{tgt}$ there exists a corresponding event in in $S_{src}$:
   
   (a) $S_{tgt}.E \subseteq \llbracket S_{src}.E \rrbracket$
   
   Function $\mu$ is injective on non-eliminated events:
   
   (b) $\forall e_1, e_2 \in S_{src}.E. \mu(e_1) = \mu(e_2) \in C \implies e_1 = e_2$

(2) Label of at least one of the events forming a redundant access pair in $S_{src}$ matches the label of its image in $S_{tgt}$:

   (a) $\forall (e_a, e_b) \in \llbracket \llbracket A \cup B \rrbracket \rrbracket ; S_{src}.po_{imm} ; \llbracket \llbracket A \cup B \rrbracket \rrbracket$

   (i) $A \neq \emptyset$ and $S_{src}.lab(e_a) = S_{tgt}.lab(\llbracket e_a \rrbracket)$, or

   (ii) $B \neq \emptyset$ and $S_{src}.lab(e_b) = S_{tgt}.lab(\llbracket e_b \rrbracket)$

   Labels of events from $C$ match in both event structures:

   (b) $\forall e \in \llbracket C \rrbracket . S_{src}.lab(e) = S_{tgt}.lab(\llbracket e \rrbracket)$

(3) Program order in $S_{tgt}$ between events from $A$ and $C$ corresponds to program order in $S_{src}$:

   (a) $[A \cup C] ; S_{tgt}.po ; [A \cup C] = [A \cup C] ; S_{src}.po \cup [A \cup C]$

   Similarly, program order in $S_{tgt}$ between events from $B$ and $C$ corresponds to program order in $S_{src}$:

   (b) $[B \cup C] ; S_{tgt}.po ; [B \cup C] = [B \cup C] ; S_{src}.po \cup [B \cup C]$

   Pair of redundant accesses in $S_{src}$ maps to the same event in $S_{tgt}$:

   (c) $[A \cup B] ; S_{src}.po_{imm} \cup [A \cup B] \subseteq 1d$

(4) Justified-from edge in $S_{src}$ to an event belonging to $A$ or $B$ corresponds either to the same event in $S_{tgt}$ or to a justified-from edge in $S_{tgt}$:

   (a) $\llbracket S_{src}.jf \rrbracket ; [A \cup B] \subseteq S_{tgt}.jf^2$

   Justified-from edge in $S_{tgt}$ to an event belonging to $A$ or $B$ corresponds either to a justified-from edge in $S_{src}$:
(b) \( S_{tgt}.jf \mid [A \cup B] \subseteq [S_{src}.jf] \)

Justified-from relation in \( S_{tgt} \) to an event belonging to \( C \) corresponds to justified-from relation in \( S_{src} \):

(c) \( S_{tgt}.jf \mid |C| = [S_{src}.jf] \mid |C| \)

(5) Equivalent-writes relation in \( S_{tgt} \) corresponds to Equivalent-writes relation in \( S_{src} \):

(a) \( S_{tgt}.ew = [S_{src}.ew] \)

(6) Coherence relation in \( S_{tgt} \) corresponds to the coherence relation in \( S_{src} \):

(a) \( S_{tgt}\cdot co = [S_{src}\cdot co] \)

(7) Extracted execution graph \( G_{src} \), corresponding to configuration \( X_{src} \), is equal to execution graph \( G_{tgt} \) modulo elimination of one of the events \( a \in A \cap G_{tgt}.E \) or \( b \in B \cap G_{tgt}.E \)

(a) \( [G_{src}.E] = G_{tgt}.E \cap S_{tgt}.E \)

Note that equations for \( 1ab, po, rf, \) and \( co \) components of the two graphs follow from the equations for event structures.

G.2.3 Adapting the proof. We need to show that simulation step respects NO-BAIT-AND-SWITCH axiom.

\[(jf \setminus (jf^2 \cup hb)) \mid po \cup ew \subseteq jf \mid (po \cup 1bpat)\]

That is, we need to show that if the simulation relation \( I(P_{src}, P_{tgt}, G_{tgt}, S_{src}, S_{tgt}, X_{src}, \mu) \) holds, the source event structure \( S_{src} \) satisfies NO-BAIT-AND-SWITCH, and the construction of the target event structure makes a step \( S_{tgt} \xrightarrow{e_t} S'_{tgt} \), s.t. \( S_{tgt} \) and \( S'_{tgt} \) both satisfy NO-BAIT-AND-SWITCH, then there exist \( S'_{src}, X'_{src}, \) and \( \mu' \) such that \( S_{src} \rightarrow^* S'_{src}, I(P_{src}, P_{tgt}, G_{tgt}, S'_{src}, S'_{tgt}, X'_{src}, \mu') \) holds, and \( S'_{src} \) also satisfies NO-BAIT-AND-SWITCH.

In order to prove this we consider each kind of elimination separately.

Store/Load: Consider the cases.

\[
\begin{align*}
\begin{array}{c}
\begin{array}{c}
a_s : W(x, a) \\
\xrightarrow{\mathcal{E}} \\
b_s : R(x, a)
\end{array} \\
(a) \text{Source event structure}
\end{array}
\quad \begin{array}{c}
\begin{array}{c}
a_t : W(x, a)
\end{array}
\end{array}
\quad \begin{array}{c}
\begin{array}{c}
b_s : R(x, a)
\end{array}
\end{array}
\end{align*}
\]

(b) Target event structure

Fig. 17. A fragment of event structure construction that justifies store/load elimination.

- \( e_t \in A \)

Let us consult Fig. 17. The target event structure adds event \( e_t = a_t : W(x, a) \). To simulate this step, the construction adds events \( a_s : W(x, a) \) and \( b_s : R(x, a) \) to the source event structure:

\[
S_{src} \xrightarrow{a_s} S'_{src} \quad b_s \xrightarrow{} S'_{src}
\]

Note that the write \( a_s \) is chosen as a justification write for read \( b_s \). Thus \( b_s \) is justified locally, i.e., from \( po \subseteq jf^2 \cup hb \) preceding write. Therefore, it is not a subject of the constraint NO-BAIT-AND-SWITCH. In other words, if \( a_s \) depends on some set of promises in \( S'_{src} \), they will be certified similarly as promises of \( a_t \) in \( S'_{tgt} \), because the set of external writes that justify the certification branch does not change.

- \( e_t \in C \)

In this case construction of the source event structure can simulate the step by adding event \( e_s \) with the same label \( S_{src} \xrightarrow{e_s} S'_{src} \). Since \( S_{tgt} \) satisfies NO-BAIT-AND-SWITCH then so \( S'_{src} \).
**Store/Load:** Consider the cases.

\[
\begin{align*}
 a_s & : W(x, \alpha) \\
 b_s & : W(x, \beta)
\end{align*}
\]

(a) Source event structure

\[
\begin{align*}
 b_t & : W(x, \beta)
\end{align*}
\]

(b) Target event structure

Fig. 18. A fragment of event structure construction that justifies store/store elimination.

- \(e_t \in A\)
  
  Let us consult Fig. 18. The target event structure adds event \(e_t = b_t : W(x, \beta)\). To simulate this step, the construction adds events \(a_s : W(x, \alpha)\) and \(b_s : R(x, \beta)\) to the source event structure:

\[
S_{src} \xrightarrow{a_s} S'_{src} \xrightarrow{b_s} S''_{src}
\]

Since \(a_s\) is a write event it is not a subject of the constraint **NO-BAIT-AND-SWITCH**. In other words, if \(b_s\) depends on some set of promises in \(S'_{src}\), they will be certified similarly as promises of \(b_t\) in \(S'_{tgt}\), because the set of external writes that justify the certification branch does not change.

- \(e_t \in C\)
  
  In this case construction of the source event structure can simulate the step by adding event \(e_s\) with the same label \(S_{src} \xrightarrow{e_s} S'_{src}\). Since \(S'_{tgt}\) satisfies **NO-BAIT-AND-SWITCH** then so \(S'_{src}\).

**Load/Load:** Consider the cases.

\[
\begin{align*}
 a_s & : R(x, \alpha) \\
 b_s & : R(x, \alpha)
\end{align*}
\]

(a) Source event structure

\[
\begin{align*}
 c_s & : W(x, \alpha)
\end{align*}
\]

(\(\cdots\))

\[
\begin{align*}
 a_t & : R(x, \alpha) \\
 c_t & : W(x, \alpha)
\end{align*}
\]

(b) Target event structure

Fig. 19. A fragment of event structure construction that justifies load/load elimination.

- \(e_t \in A\)
  
  Let us consult Fig. 19. The target event structure adds event \(e_t = a_t : R(x, \alpha)\). Let \(c_t : W(x, \alpha)\) be a write event chosen to justify \(a_t\). To simulate this step, the construction adds events \(a_s : R(x, \alpha)\) and \(b_s : R(x, \alpha)\) to the source event structure:

\[
S_{src} \xrightarrow{a_s} S'_{src} \xrightarrow{b_s} S''_{src}
\]

As a justification write for both events the construction takes write event \(c_s\), s.t. \(\mu(c_s) = c_t\). It implies that \(b_s\) is justified locally, i.e., from \(jf : po \subseteq jf^{\Omega} : hb\) preceding write. Therefore, it is not a subject of the constraint **NO-BAIT-AND-SWITCH**. In other words, if \(a_s\) depends on some set of promises in \(S'_{src}\), they will be certified similarly as promises of \(a_t\) in \(S'_{tgt}\), because the set of external writes that justify the certification branch does not change.

- \(e_t \in C\)
  
  In this case construction of the source event structure can simulate the step by adding event \(e_s\) with the same label \(S_{src} \xrightarrow{e_s} S'_{src}\). Since \(S'_{tgt}\) satisfies **NO-BAIT-AND-SWITCH** then so \(S'_{src}\).