Model Checking on Multi-execution Memory Models

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

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

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

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

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

ACM Reference Format:
Evgenii Moiseenko, Michalis Kokologiannakis, and Viktor Vafeiadis. 2022. Model Checking on Multi-execution Memory Models. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 152 (October 2022), 70 pages. https://doi.org/10.1145/3563315

1 INTRODUCTION

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

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

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

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

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

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

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

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

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

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

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

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

2 OVERVIEW

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

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

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

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

2.1 Execution Graphs and Data Race Freedom

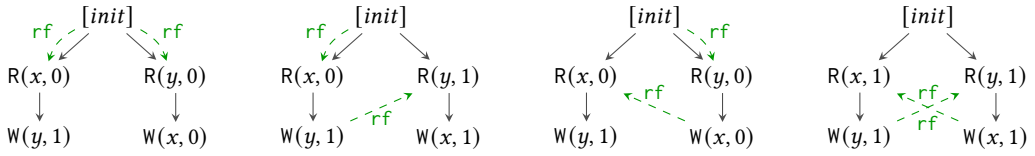
In the literature of axiomatic (a.k.a. declarative) per-execution memory models, the possible executions of a program P under a model M are represented as a set of *execution graphs* that correspond to the instructions of P and satisfy M's consistency predicate. Execution graphs consist of:

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

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

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

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



(a) Acyclic execution (b) Acyclic execution (c) Acyclic execution (d) Cyclic execution

Fig. 1. Execution graphs of LB.

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

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

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

2.2 Load Buffering Race Freedom

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

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

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

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

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

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

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

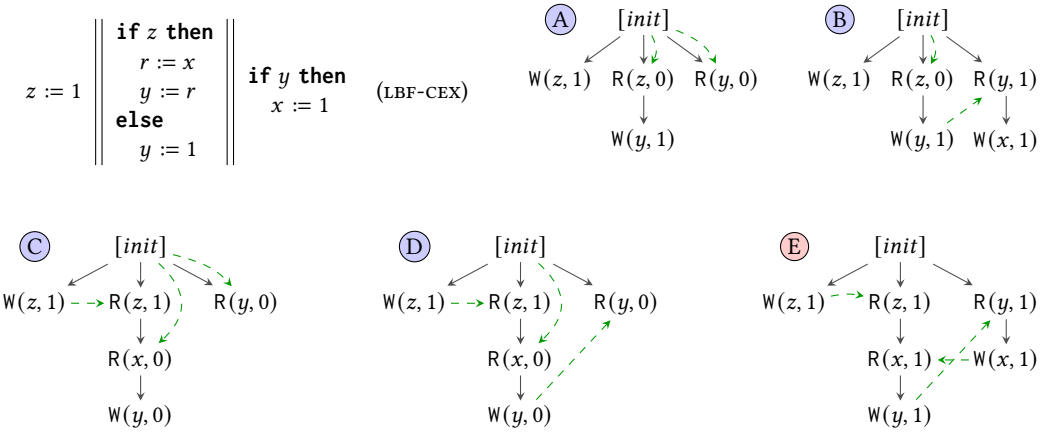


Fig. 2. A program with its RC11-consistent execution graphs and a problematic cyclic execution.

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

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

Theorem 2.5. IMM *satisfies* $\mathsf{LBRF}_{\mathsf{RC11}}$. *(See 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 $\mathsf{LBRF}_{\mathsf{RC11}}$. To see this, consider the lbf-cex program along with its executions shown in Fig. 2. Although none of its RC11-consistent executions (A, B, C, D) contains an LB race, both Promising and Weakestmo allow the additional execution E where r gets the value 1. This execution can arise in the following manner. First, thread 2 promises the $\mathtt{W}(y, 1)$ store; the promise is allowed because thread 2 can read $z = 0$ and fulfill it. Then, thread 3 executes: it reads $y = 1$ and writes 1 to x. Finally, threads 1 and 2 execute: thread 2 reads $z = 1$ and $x = 1$, and subsequently writes 1 to y thereby fulfilling its promise.

That said, it is fairly straightforward to strengthen Promising and Weakestmo to satisfy LBRF by restricting when promises can be made. The existing condition of Promising requires there to be a thread-local execution that certifies the promise. In addition to that, we can require there to be an execution witnessing an LB race. As we shall shortly see, however, satisfying LBRF alone is not

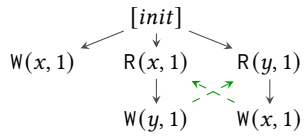
sufficient for effective model checking. We need another locality property that completely forbids non-local certifications that depend on external writes like the $z = 1$ write in the example above.

2.3 Certification Locality

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

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

$$x := 1 \;\Big\|\; \begin{array}{l} r_1 := x \;\; /\!/ \textit{reads 1} \\ y := r_1 \end{array} \;\Big\|\; \begin{array}{l} r_2 := y \;\; /\!/ \textit{reads 1} \\ x := r_2 \end{array}$$



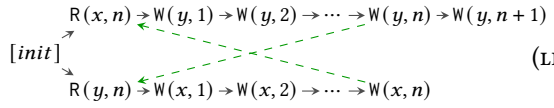
(LB+DEP+W)

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

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

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

$$\begin{array}{l} r_1 := x \\ \textbf{for } i := 0 \textbf{ to } r_1 \\ \quad y := i + 1 \end{array} \;\Big\|\; \begin{array}{l} r_2 := y \\ \textbf{for } j := 1 \textbf{ to } r_2 \\ \quad x := j \end{array}$$



(LB+LOOP)

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

2.3.1 Enforcing CL. Repairing Promising to satisfy CL is not as straightforward as it is for LBRF because it does not track the exact set of writes read by a thread. We therefore instrument the Promising state by attaching to each promise a set of external writes that must be read before fulfilling the promise. We then check that as long as a thread contains outstanding promises, it can only read from external writes that are recorded in its promise set and, moreover, that when a promise is fulfilled all its attached external writes have been read. We present a suitable definition in 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 $\text{LBRF}_{\text{RC11}}$ and DRF_{SC} guarantees (§3.3), its expected compilation schemes to hardware models are sound (§3.4), and it supports the expected local reorderings and eliminations (§3.5).

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

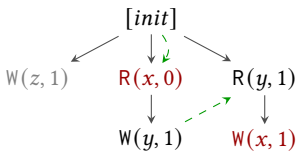
Apart from serving as criteria to rule out certain weak behaviors of multi-execution memory models, LBRF and CL are instrumental in making model checking of programs on such models feasible. We briefly describe how WMC, our model checking algorithm for Weakestmo2, exploits these properties. (We defer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

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

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

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

$$z := 1 \;\Big\|\; \begin{array}{l}\textbf{if } x \textbf{ then}\\ \quad r := z\\ \quad y := r\\ \textbf{else}\\ \quad y := 1\end{array} \;\Big\|\; \begin{array}{l}\textbf{if } y \textbf{ then}\\ \quad x := 1\end{array}$$



(LB+RDEP)

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

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

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

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

3 REPAIRING WEAKESTMO

In this section, we describe how Weakestmo2, our strengthening of Weakestmo, supports LBRF and CL. In what follows, we assume a simplified version of the model, containing only relaxed accesses and fences. For the full model, we refer the reader to Chakraborty et al. [2019] and 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 $\langle \mathtt{id}, \mathtt{tid}, \mathtt{lab} \rangle$ where $\mathtt{id} \in \mathbb{N}$ is a unique identifier for the event, $\mathtt{tid} \in \mathbb{N}_\bot$ identifies the thread to which the event belongs ($\bot$ for initialization events), and $\mathtt{lab}$ is a *label* of the form: (1) $\mathtt{R}(x, v)$ for a read of $v \in \mathsf{Val}$ from $x \in \mathsf{Loc}$; (2) $\mathtt{W}(x, v)$ for a write of $v \in \mathsf{Val}$ to $x \in \mathsf{Loc}$; (3) $\mathtt{F}$ for a fence.

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

Further, given a relation r, we use $dom(r)$ and $rng(r)$ to denote its domain and codomain, while $r^?$, r^+, and r^* denote its reflexive, transitive, and reflexive-transitive closures, respectively. We write

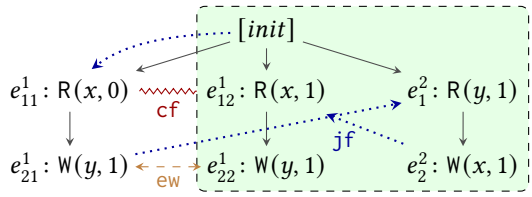


Fig. 3. LB event structure and an extracted execution.

r^{-1} for the inverse of r, $[A]$ for the identity relation $\{(a,a) \mid a \in A\}$, $r_1; r_2$ for the composition of r_1 and r_2: $\{(a,b) \mid \exists c\ (a,c). \in r_1 \wedge (c,b) \in r_2\}$, and $r|_s$ for the restriction of r onto set s: $r|_s \triangleq r \cap s \times s$. When $\mathtt{f}$ is a function, we define $=_{\mathtt{f}} \triangleq \{(a,b) \mid \mathtt{f}(a) = \mathtt{f}(b) \neq \bot\}$.

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

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

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

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

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

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

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

Next, we discuss how to extract an execution graph from an event structure. To do so, it is not sufficient to just take a conflict-free subset of events. For example, taking the single event $\{e^1_{12}\}$

from the structure in Fig. 3 does not form an execution graph corresponding to the program LB. In addition, we must take all po-predecessors of the event and the writes that justify them. There is, however, one subtle point. Event e_1^2 is justified by e_{21}^1, which is in conflict with e_{12}^1; thus, instead of e_{21}^1, we pick the equivalent write e_{22}^1. To achieve that, we define the derived *reads-from* relation $\mathtt{rf} \triangleq (\mathtt{ew}^* ; \mathtt{jf}) \setminus \mathtt{cf}$ by extending the jf relation to $\mathtt{ew}^*$ equivalence classes.

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

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

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

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

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

3.1 Weakestmo **Consistency**

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

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

Consider Fig. 3 again. Write event e_{21}^1 is a pending promise w.r.t. e_{12}^1 and a certified promise w.r.t. e_{22}^1. 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.

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

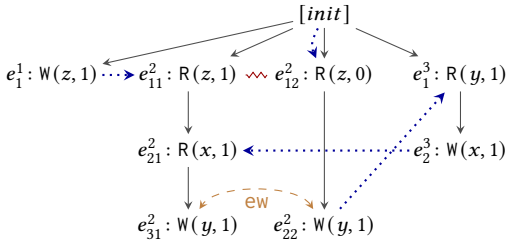


Fig. 4. Weakestmo2 **inconsistent** event structure of LBF-CEX.

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

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

3.2 Weakestmo2 **Consistency**

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

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

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

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

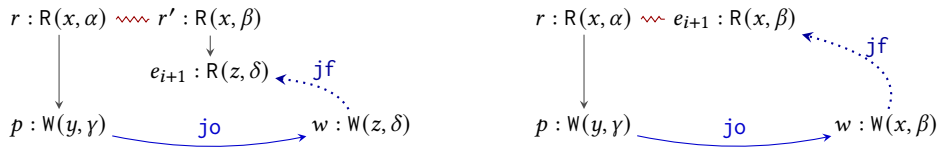
- $(\mathtt{jf} \setminus (\mathtt{jf}^? ; \mathtt{hb})) ; \mathsf{po} ; \mathsf{ew} \subseteq \mathtt{jf} ; (\mathsf{po} \cup \mathtt{lbpat})$ (NO-BAIT-AND-SWITCH)

where $\mathtt{lbpat} \triangleq \mathtt{cf}_{\mathtt{imm}} ; [rng(\mathtt{jf} \cap (\mathtt{jf}^? ; \mathtt{hb}))] ; \mathsf{po}$ 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.



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

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

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

Due to this axiom, the event structure of Fig. 4 is not Weakestmo2-consistent because event e^3_2 externally justifies e^2_{21} but does justify any event po-before e^2_{22}. In contrast, the event structure of Fig. 3 is Weakestmo2-consistent. The only relevant external justification is that of e^1_{12} by e^2_2, which is allowed because e^1_{12} is in immediate conflict with e^1_{11}, which is po-before e^1_{21}.

3.3 Load Buffering and Data Race Freedom

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

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

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

THEOREM 3.12. Weakestmo2 *satisfies* $\mathsf{LBRF}_{\mathsf{RC11}}$. *(See Appendix D for the full proof.)*

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

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

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

This situation is depicted in Fig. 5a. We have a newly added read event e_{i+1} and a promise p. Read events r and r' are the first events at which two branches of the event structure diverge and become conflicting. The constraint NO-BAIT-AND-SWITCH guarantees that the read e_{i+1} cannot observe promise p in the middle of the certification branch, thus it has to be that $e_{i+1} = r'$ (see Fig. 5b). Then it is easy to see that events r and w form a load-buffering race. Both of these events belong to the event structure S_i obtained on the previous step. Because of our inductive assumption, S_i is promise-free. Thus we can extract an RC11-consistent execution containing a load-buffering race,

contradicting our assumption that P is LB-race-free under RC11. Therefore, it has to be that S_{i+1} remains promise-free. □

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

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

3.4 Soundness of Compilation Mappings

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

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

Theorem 3.14. *Let P be a program, and G be an* IMM*-consistent execution graph of P. Then, there exists an* Weakestmo2*-consistent event structure S of P such that $S \triangleright G$.*

(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 $G \vdash TC \xrightarrow{e} TC'$ where TC and TC' are *traversal configurations* and e is an event being traversed. Traversal configuration is a tuple $\langle C, I\rangle$, where $C \subseteq G.\mathsf{E}$ is a set of *covered events* and $I \subseteq G.\mathtt{W}$ is a set of *issued writes*.

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

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

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

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

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

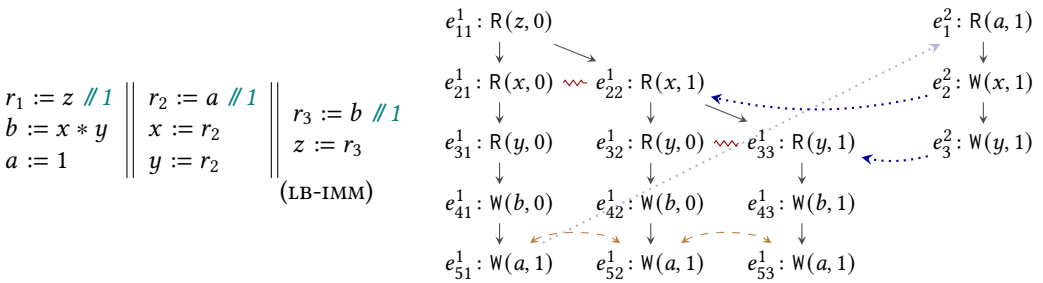


Fig. 6. LB-IMM and its partial Weakestmo2-consistent event structure.

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

Subsequently, the traversal issues $\mathtt{W}(b,1)$. At this point, the construction of Moiseenko et al. [2020] adds a branch $e^1_{22} \to e^1_{33} \to e^1_{43} \to e^1_{53}$. It does so by picking suitable justification writes *for all reads* on which the issued write $\mathtt{W}(b,1)$ 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 $G.\mathsf{ppo}$-preceding reads *incrementally* one-by-one, constructing a series of certification branches, as shown in Fig. 6. That is, we first construct the intermediate branch $e^1_{22} \to e^1_{32} \to e^1_{42} \to e^1_{52}$. Doing so satisfies NO-BAIT-AND-SWITCH because the new branch differs from the previous branch only at the point of immediate conflict: $e^1_{21} \rightsquigarrow e^1_{22}$. Starting from this intermediate branch, it becomes possible to add the required branch $e^1_{33} \to e^1_{43} \to e^1_{53}$ which now has the event e^1_{43} matching the issued write $\mathtt{W}(b,1)$.

The remaining part of the simulation process is not shown in Fig. 6 because it proceeds unchanged compared to Moiseenko et al. [2020]: first $\mathtt{R}(b,1)$ is covered, then $\mathtt{W}(z,1)$ 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 $P_{\mathtt{src}}$ to a target program $P_{\mathtt{tgt}}$ is *sound* if does not add any new behaviors. That is, for every Weakestmo2-consistent execution $G_{\mathtt{tgt}}$ of $P_{\mathtt{tgt}}$, there exists some Weakestmo2-consistent execution $G_{\mathtt{src}}$ of $P_{\mathtt{src}}$ with the same behavior: $\mathcal{B}(G_{src}) = \mathcal{B}(G_{\mathtt{tgt}})$.

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* $\mathsf{a} = (r_1 := x)$ *and* $\mathsf{b} = (y := r_2)$ *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 $G_{\mathtt{tgt}}$ of $P_{\mathtt{tgt}}$, we consider event structure $S_{\mathtt{tgt}}$, s.t. $S_{\mathtt{tgt}} \triangleright G_{\mathtt{tgt}}$, and we built it

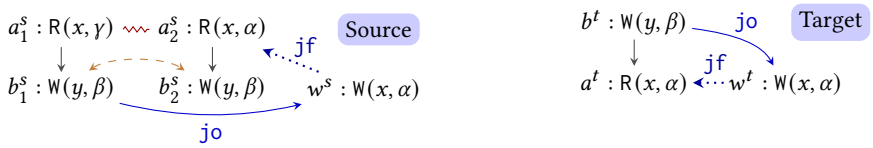


Fig. 7. A fragment of the event structure construction that justifies load/store reordering.

incrementally from the initial event structure: $S_{\text{init}}(P_{\mathtt{tgt}}) \rightarrow^* S_{\mathtt{tgt}}$. Following these steps, we will construct $S_{\mathtt{src}}$ and the required graph $G_{\mathtt{src}}$.

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

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

The construction of Chakraborty et al. [2019, §F] therefore creates an auxiliary execution branch in the source event structure consisting of events a_1^s and b_1^s. The construction justifies the $a_1^s : \mathtt{R}(x,\gamma)$ event by choosing the $S_{\mathtt{src}}.\mathtt{co}$-maximal non-conflicting write from $S_{\mathtt{src}}.\mathtt{jf}^?$; $S_{\mathtt{src}}.\mathtt{hb}$ prefix of a_1^s. Since instructions a and b are assumed to be independent, the choice of the value γ for the read a_1^s cannot affect the value of the write $b_1^s : \mathtt{W}(y,\beta)$.

The construction then proceeds by adding events to the target and source event structures until the target reaches the event a^t. At this point, the construction adds another branch to the source consisting of events a_2^s and b_2^s. Now, the event a_2^s can have the required label $\mathtt{R}(x,\alpha)$ because there is already a justifying event w^s in the source event structure matching the target's justification event w^t. Finally, the write b_2^s is announced to be equal to b_1^s.

What remains to be shown is that the two conflicting branches $a_1^s \rightarrow b_1^s$ and $a_2^s \rightarrow b_2^s$ satisfy the axiom NO-BAIT-AND-SWITCH. Indeed, they only differ at the point of the immediate conflict, and moreover, the read a_1^s is justified from a $(S_{\mathtt{src}}.\mathtt{jf}^?$; $S_{\mathtt{src}}.\mathtt{hb})$-preceding write. □

4 WMC: WEAKESTMO2 MODEL CHECKING

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

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.)

Algorithm 1 Main exploration algorithm

```
1:  procedure Visit(P, G, Π)
2:    if ¬cons_{RC11\porf}(G) then return
3:    switch a ← next_{P,Π}(G) do
4:      case a = ⊥
5:        if Π = ∅ then output "Exec OK"
6:      case a ∈ error
7:        exit("Erroneous execution")
8:      case a ∈ R
9:        for w ∈ GetRFs(G, Π, a) do
10:         Visit(P, SetRF(G, w, a), Π)
11:     case a ∈ W
12:       Π' ← {⟨w, G_w⟩ ∈ Π | w ≠ a}
13:       Visit(P, G, Π')
14:       VisitRevisits(P, G, Π, Π', a)
15:     otherwise Visit(P, G, Π)

1:  function GetRFs(G, Π, r)
2:    W ← G.W_{loc(r)}
3:    if Π ≠ ∅ then
4:      L ← {w' | ∃⟨w, G'⟩ ∈ Π.
                ⟨w', w⟩ ∈ G'.rf^?; G.hb}
5:      W ← W ∩ L
6:    return W
```

Algorithm 2 Exploration of revisits

```
1:  procedure VisitRevisits(P, G, Π, Π', a)
2:    if Π' ≠ ∅ then ⟨Rs, Π_c⟩ ← ⟨∅, ∅⟩          ▷ In cert
3:    else if Π ≠ ∅ then                          ▷ Cert OK
4:      ⟨Rs, Π_c⟩ ← ⟨CertRevs(G, a), {a}⟩
5:    else                                        ▷ Normal exec
6:      ⟨Rs, Π_c⟩ ← ⟨GetRevs(G, a), ∅⟩
7:    for ⟨w, r⟩ ∈ Rs do
8:      G' ← Restrict(G, r, w)\rng([r]; G.po)
9:      Π' ← Promises(G, r, {w} ∪ Π_c ∪ InCyc(G, r))
10:     Visit(P, G', Π')

1:  function GetRevs(G, a)
2:    return {⟨a, r⟩ | r ∈ (G.T_{loc(a)} \ dom(G.porf; [a]))
                       ∪ dom(G.lbrace ; [a])}

1:  function Promises(G, r, S)
2:    return {⟨w', G⟩ | ⟨r, w'⟩ ∈ G.po ; [dom(G.rfe)]
                        ∧ w' ∈ dom(G.porf; [S])}

1:  function CertRevs(G, e)
2:    return {⟨r, w⟩ | ⟨r, e⟩ ∈ po ∧ G.rf; [r] ⊆ G.rf^?; G.hb
                     ∧ w ∈ G.W_{loc(r)}} ∪
              ⋃_{w ∈ dom([W];G.po;[e])} GetRevs(G, T, w)
```

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

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

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

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

If a is a write, apart from simply recursing further (Line 13), Visit has to also check whether w can *revisit* any of the existing reads in G. This is necessary because, when a read r is added to the graph, it may well be the case that some write from which r could also read from has not yet been added to the graph. Thus, whenever Visit adds a write a, it also checks whether any of the existing reads can be revisited to read from a, and explores these options via VisitRevisits (Line 14).

Otherwise, Visit simply recurses further.

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

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

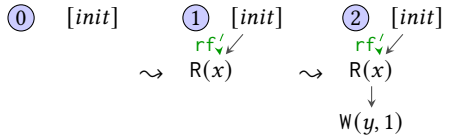
Subsequently (and in accordance to standard DPOR approaches), for each such revisit pair $\langle r, w \rangle$, VisitRevisits first restricts G (Line 8) so that it only contains the events that were added before r, as well as the events that are porf-before w (as w was added after r). Analogously, it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

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

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

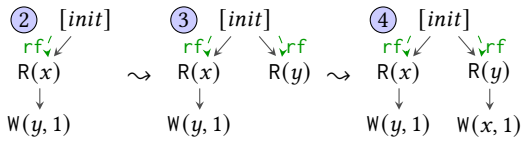
We choose this example because it has 3 consistent executions under RC11 (Figures 1a to 1c) and one additional execution under Weakestmo2. Running GenMC on LB-NODEP highlights the difficulties arising when trying to enumerate the executions of programs with porf cycles. In the presentation below, we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

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



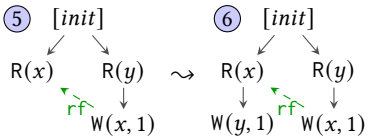
Note that, because there are no other reads-from options for $\mathsf{R}(x)$ except for 0, the loop in Line 9 in Visit will only add one child node to the recursion tree. Similarly, because there are no revisit options for $\mathsf{W}(y, 1)$, VisitRevisits is not executed, and GenMC only calls Visit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

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



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

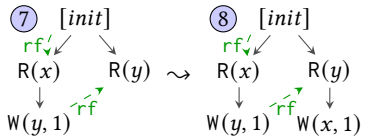
In the non-revisiting case, the graph is complete (corresponding to Fig. 1a), so let us focus on the revisiting case. The graph created for the revisiting exploration is graph 5 below.



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

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

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



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

4.2 WMC: Model Checking under Weakestmo2

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

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

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

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

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

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

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

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

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

As a second step, whenever Visit encounters a non-empty promise set, WMC initiates a *certification phase*. During this phase, WMC operates in a restricted mode: all promises of Π have

to be fulfilled (i.e., the corresponding writes have to be re-added to the graph), and all non-local explorations (see §2.3) are postponed until the certification phase succeeds.

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

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

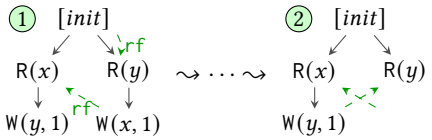
Second, the GetRFs function is modified to constrain the possible reads-from options during the certification phase (Line 3). The modified version of the function calculates the set of local writes that were used to issue promises (Line 4). Reading from these writes is safe, since it cannot lead to bait-and-switch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

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

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

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

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



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

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

We conclude this example with two remarks.

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.

$$\begin{array}{l} r_1 := x \\ \textbf{if } r_1 = 0 \textbf{ then} \\ \quad y := 1 \end{array} \;\Big\|\; \begin{array}{l} r_2 := y \\ \textbf{if } r_2 = 0 \textbf{ then} \\ \quad x := 1 \end{array} \quad \text{(LB+CTRL)} \qquad \begin{array}{l} r_1 := x \\ y := 1 \end{array} \;\Big\|\; \begin{array}{l} r_2 := z \\ r_3 := y \\ x := 1 \end{array} \;\Big\|\; z := 2 \quad \text{(RLB+W)}$$

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

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

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

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

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

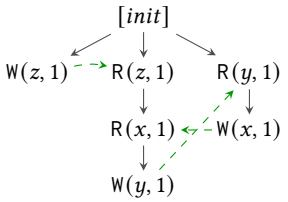
Continuing with the calculation of the promise set, given a revisit $\langle w, r\rangle$, the definition of Promises simply returns the writes after po-after r that are read externally, and are porf-before its last parameter S.

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

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

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

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



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

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

5 EVALUATION

We evaluate WMC by answering the following questions:

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

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

- GenMC [Kokologiannakis et al. 2019; 2021] is the stateless model checker that we build upon. It supports the RC11 memory model, which does not permit LB behaviors.
- HMC [Kokologiannakis et al. 2020] is an extension of GenMC that verifies programs under IMM [Podkopaev et al. 2019], a weak memory model that allows certain LB behaviors by keeping track of dependencies between instructions and forbidding cycles consisting solely of dependencies and `rf` edges.
- HMC_{LBRF} is a variant of HMC that we implemented in order to further demonstrate the benefits of LBRF. HMC_{LBRF} leverages LBRF and starts calculating dependencies only when it finds a load buffering race.
- Nidhugg [Abdulla et al. 2015a; 2016] is a stateless model checker that, among others, supports the POWER memory model by tracking dependencies (similarly to HMC).
- CDSChecker [Norris et al. 2013] verifies programs under an informally defined strengthening of the original C11 model, which forbids thin-air behaviors using a notion of promises.
- 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, 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 256GB of RAM, running a custom Debian-based distribution. We used LLVM 7 for HMC (v0.5), GenMC (v0.5), and Nidhugg (v0.3), commit #da671f7 for CDSChecker, and commit #85c8130 for RMEM (v0.1). All reported times are in seconds, unless explicitly noted otherwise. We set the timeout limit to 30 minutes.

5.1 Load Buffering Races and WMC's Overhead

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

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

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

Table 1. Overhead of LBRF on GenMC benchmarks

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

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

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

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

5.2 Comparison with Other Model Checkers

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

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).

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

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

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

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

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

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

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

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

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

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

5.3 Load-Buffering Benchmarks

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

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

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

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

Table 4. Load buffering benchmarks

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

5.4 Automatically Fixing Load-Buffering Races

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

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

6 RELATED WORK AND CONCLUSION

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

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

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

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

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

ACKNOWLEDGMENTS

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

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis (2020). "Reconciling Event Structures with Modern Multiprocessors." In: *ECOOP 2020*. Vol. 166. Leibniz International Proceedings in Informatics

(LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 5:1–5:26. DOI: 10.4230/LIPIcs.ECOOP.2020.5.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli (2015). "Common compiler optimisations are invalid in the C11 memory model and what we can do about it." In: *POPL 2015*. Mumbai, India: ACM, pp. 209–220. DOI: 10.1145/2676726.2676995.

A PROOF OF THE LBRF THEOREM FOR IMM

In this section, we prove that IMM [Podkopaev et al. 2019] provides the $\text{LBRF}_{\text{RC11}}$ 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 $\langle \texttt{data}, \texttt{ctrl}, \texttt{addr}, \texttt{casdep} \rangle$. The dependency relation is the union of the above:

$$\texttt{deps} \triangleq \texttt{data} \cup \texttt{ctrl} \cup \texttt{addr} ; \text{po}^? \cup \texttt{casdep}$$

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

$$\text{bob} \triangleq \text{po} ; [\text{W}^{\texttt{rel}}] \cup [\text{R}^{\texttt{acq}}] ; \text{po} \cup \text{po} ; [\text{F}] \cup [\text{F}] ; \text{po} \cup [\text{W}^{\texttt{rel}}] ; \text{po} \cap =_{\texttt{loc}} ; [\text{W}]$$

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

$$\text{ppo} \triangleq [\text{R}] ; (\texttt{deps} \cup \texttt{rfi})^+ ; [W]$$

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

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

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

$$\text{ar} \triangleq \texttt{rfe} \cup \text{bob} \cup \text{ppo} \cup \texttt{detour} \cup \text{psc}_{\text{F}}$$

Similarly to Weakestmo and RC11, the IMM model also defines the relations sw, hb, scb, psc_{F}, psc_{base}, and psc. However, the IMM has a slightly different definition of 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 that 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:

- ar is acyclic (IMM-NO-THIN-AIR)
- $\text{hb}_{\text{IMM}} ; \texttt{eco}^?$ is irreflexive. (IMM-COHERENT)
- $\texttt{rmw} \cap (\texttt{fr} ; \texttt{co}) = \emptyset$. (RMW-ATOMIC)
- psc is acyclic. (IMM-SEQUENTIAL-CONSISTENCY)

Definition A.7 (porf-sequence). Given an execution graph G, a set of events $\mathcal{E} \subseteq G.\text{E}$ is a *porf-sequence* if there exists a linearization $\{e_1, \ldots, e_n\} = \mathcal{E}$, 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:

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

Definition A.8 (porf-cycle). A set of events $\mathcal{E} \subseteq G.\text{E}$ is a *porf-cycle* if it is a closed porf-sequence, i.e., $\langle e_n, e_1 \rangle \in \text{po}_{\texttt{imm}} \cup \texttt{rfe}$.

Definition A.9 (porf-span). Let $\mathcal{E}$ be a porf-cycle of graph G. Its *span* is the subsequence $\mathcal{S} = \mathcal{E} \cap (dom(G.\mathtt{rfe} \,;\, [\mathcal{E}]) \cup rng([\mathcal{E}] \,;\, G.\mathtt{rfe}))$. 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 $\mathcal{E}$ be a porf-cycle, s.t. the porf-prefix of the cycle does not contain other cycles, i.e., for $X = dom(G.\mathtt{porf} \,;\, [\mathcal{E}]) \setminus \mathcal{E}$ the graph $G|_X$ is porf-acyclic. Then, there exists a RC11-consistent execution graph G' of P with an LB race.*

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

Second, the cycle $\mathcal{E}$ 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 porf acyclic RC11 consistent graph with LB race.

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

- $r \in G.\mathtt{R}^{\mathtt{rlx}} \setminus G.\mathtt{R}_{\mathtt{ex}}$ and $w \in G.\mathtt{W}^{\mathtt{rlx}}$
- $\langle r, w\rangle \in G.\mathtt{po} \setminus (G.\mathtt{ppo} \cup G.\mathtt{bob})$
- $dom([G.\mathtt{R}_{G.\mathtt{loc}(r)} \cap \mathcal{S}] \,;\, G.\mathtt{po} \,;\, [r])$ is empty
- $\langle w', r\rangle \in G.\mathtt{rfe}$
- $\langle w', r\rangle \notin \mathtt{hb}_{\mathsf{IMM}} \cup G.\mathtt{hb}^{-1}_{\mathsf{IMM}}$

We do this by induction on the length n of the span $\mathcal{S}$.

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 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 ppo, bob and rfe edges, that is the following is false:

- $\langle e_i, e_{i+1}\rangle \in \mathtt{ppo} \cup \mathtt{bob} \cup \mathtt{rfe}$ for all $1 \le i < n$;
- $\langle e_n, e_1\rangle \in \mathtt{ppo} \cup \mathtt{bob} \cup \mathtt{rfe}$.

Otherwise we would get the ppo ∪ bob ∪ rfe cycle, which is forbidden by IMM consistency.

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

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

Second, suppose there exists $c \in G.\mathtt{R}_{G.\mathtt{loc}(a)} \cap \mathcal{S}$, s.t. $\langle c, a\rangle \in G.\mathtt{po}$. Then we can construct a span of smaller length with an edge $\langle c, b\rangle$ which, 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 $\mathcal{S}$ s.t. $\langle w_a, a\rangle \in G.\mathtt{rfe}$. Suppose $\langle w_a, a\rangle \in \mathtt{hb}_{\mathsf{IMM}} \cup \overline{G}.\mathtt{hb}^{-1}_{\mathsf{IMM}}$. If $\langle w_a, a\rangle \in \mathtt{hb}_{\mathsf{IMM}}$ then consider the last edge in $\mathtt{hb}_{\mathsf{IMM}}$ path. It cannot be $\mathtt{sw}_{\mathsf{IMM}}$ edge since it would imply $a \in G.\mathtt{R}^{\sqsupseteq \mathtt{acq}}$ which cannot be true as we have shown above. If the last edge is po edge, then there should exists $c \in rng([w_a] \,;\, G.\mathtt{rfe})$ and $\langle c, a\rangle \in G.\mathtt{po}$. But we have proven there exists no such c. Next assume $\langle a, w_a\rangle \in \mathtt{hb}_{\mathsf{IMM}}$. Then there should exists $d \in \mathtt{W}$, s.t. $\langle a, d\rangle \in \mathtt{po}$, 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 $\mathcal{S}$, s.t. $\langle d, d'\rangle \in (\mathsf{po} \cap =_{\mathsf{loc}})^?$. Then again we can construct a smaller span with an edge $\langle a, d'\rangle$ and by induction get suitable edge $\langle r, w\rangle$. If neither $\langle w_a, a\rangle \in \mathsf{hb}_{\mathsf{IMM}}$ nor $\langle a, w_a\rangle \in \mathsf{hb}_{\mathsf{IMM}}$, we have $r = a$, $w = b$ and $w' \in dom(G.\mathtt{rfe} ; [a])$ which satisfy our constraints.

Otherwise, let, for example $a \in G.\mathsf{R}$ and $b \in G.\mathsf{R}$ 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 $\mathcal{S}$ to be closed, there should exist $w', r' \in \mathcal{S}$, s.t. $\langle w', r'\rangle \in G.\mathtt{rfe}$, $\langle b, w'\rangle \in \mathsf{po}$, and $\langle r', w_b\rangle \in G.\mathtt{porf}$. 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 $\langle r, w\rangle$. Other cases for a and b can be handled similarly.

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

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

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

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 $\langle r, w\rangle \notin G.\mathtt{ppo}$, the label of w would not be affected if we'll redirect the read r to read-from $G.\mathtt{rfe}^? ; G.\mathsf{hb}_{\mathsf{IMM}}$ prior write. Redirecting to such write is safe, since it cannot violate axiom IMM-COHERENT. Also, because $r \notin G.\mathsf{R}_{\mathsf{ex}}$, 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 $\langle r, w\rangle$ and a write event w', s.t.:

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

That is, the pair $\langle r, w'\rangle$ forms an LB race. Since the sequence $\mathcal{E}$ does not form a cycle in $\overline{G}$, and its porf prefix $\mathcal{P} \triangleq dom(\overline{G}.\mathtt{porf} ; [\mathcal{E}])$ also contains no cycles by our assumption, we have that $\overline{G}|_{\mathcal{P}}$ 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 $\mathsf{hb}_{\mathsf{RC11}} \subseteq \mathsf{hb}_{\mathsf{IMM}}$). Therefore $\overline{G}$ is a RC11-consistent graph of P and $\langle r, w'\rangle$ is an LB race. □

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 $G.\mathtt{porf}$ cycles. Then we can order non-overlapping $G.\mathtt{porf}$ cycles: the cycle $\mathcal{E}_1$ is said to be less than $\mathcal{E}_2$ if $\mathcal{E}_1 \subseteq dom(G.\mathtt{porf}\ ;\ [\mathcal{E}_2])$. It is easy to show that this definition gives us partial order. Next pick an arbitrary minimal cycle $\mathcal{E}$. Because its minimal w.r.t. to the partial order defined above, its $G.\mathtt{porf}$ prefix cannot contain another $G.\mathtt{porf}$ 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 $\mathcal{TS}$, 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, σ, 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:

$$\frac{TS = \mathcal{TS}(i) \qquad \langle TS, M\rangle \to \langle TS', M'\rangle \qquad \langle TS', M'\rangle \xrightarrow{\text{promise-free}}{}^{*} \langle TS'', M''\rangle \qquad TS''.\mathsf{prm} = \emptyset}{\langle \mathcal{TS}, M\rangle \to \langle \mathcal{TS}[i \mapsto TS'], M'\rangle} \text{ (Machine Step)}$$

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, $\sigma \xrightarrow{\ell} \sigma'$.

$$\frac{\sigma \xrightarrow{\epsilon} \sigma'}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma', V, P\rangle, M\rangle} \text{ (Silent)}$$

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.

$$\frac{\sigma \xrightarrow{\mathsf{R}(x,v)} \sigma' \qquad V(x) = t \qquad \langle x : v@t\rangle \in M}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma', V, P\rangle, M\rangle} \text{ (Read-Local)}$$

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.

$$\frac{\sigma \xrightarrow{\mathtt{R}(x,v)} \sigma' \qquad V(x) < t \qquad \forall p \in dom(P).\ m \in P(p) \qquad m = \langle x : v@t \rangle \in M \qquad P' = \{\langle p, \mathcal{W} \setminus \{m\}\rangle \mid \langle p, \mathcal{W}\rangle \in P\}}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma', V[x \mapsto t], P'\rangle, M\rangle} \text{ (Read-External)}$$

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.

$$\frac{M' = M \uplus \{m\} \qquad P' = P \uplus \{\langle m, \mathcal{W}\rangle\}}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma, V, P'\rangle, M'\rangle} \text{ (Promise)}$$

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.

$$\frac{\sigma \xrightarrow{\mathtt{W}(x,v)} \sigma' \qquad V(x) < t \qquad \langle m, \emptyset\rangle \in P \qquad m = \langle x : v@t\rangle \qquad P' = P \setminus \{\langle m, \emptyset\rangle\}}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma', V[x \mapsto t], P'\rangle, M\rangle} \text{ (Fulfill)}$$

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.

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

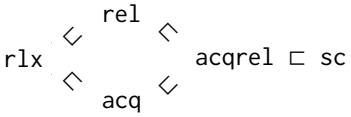
Promise-free thread transitions are all the above, except for the (Extend-Promise) step. It can be shown that if there is a sequence of thread-local steps that reaches a configuration without any promises, that sequence can be rearranged so that every (Promise) step is immediately before the corresponding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

C COMPLETE DEFINITION OF WEAKESTMO2

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

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

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



Definition C.2. An *event structure* is S is a tuple $\langle \mathsf{E}, \mathsf{po}, \mathtt{jf}, \mathsf{ew}, \mathtt{co}\rangle$ where:

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

Definition C.3. *Conflict* relation is defined as follows: $\mathtt{cf} \triangleq =_{\mathtt{tid}} \setminus (\mathsf{po} \cup \mathsf{po}^{-1})^?$

Definition C.4. *Immediate program order* and *immediate conflict* relations are defined as follows:

$$\mathsf{po}|_{\mathsf{imm}} \triangleq \mathsf{po} \setminus (\mathsf{po} \,;\, \mathsf{po})$$
$$\mathtt{cf}|_{\mathsf{imm}} \triangleq \mathtt{cf} \setminus (\mathtt{cf} \,;\, \mathsf{po} \cup \mathsf{po}^{-1} \,;\, \mathtt{cf})$$

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

(1) There exists the initialization event $e_0 \in \mathsf{E}$.
We assume that: $e_0 = \langle 0, \bot, \mathsf{W}^{\mathsf{rlx}}(\bot, 0)\rangle$ For $\mathtt{f} \in \{\mathtt{tid}, \mathtt{lab}, \mathtt{loc}\}$ we define:
$$<_{\mathtt{f}} \triangleq \{e_0\} \times (\mathsf{E} \setminus \{e_0\})$$
$$\leq_{\mathtt{f}} \triangleq \{e_0\} \times (\mathsf{E} \setminus \{e_0\}) \cup =_{\mathtt{f}}$$
(2) po is strict partial order.
(3) po orders events from the same thread (the initialization event is placed before other events):
$$<_{\mathtt{tid}} \subseteq \mathsf{po} \subseteq \leq_{\mathtt{tid}}$$
(4) po is downward-total: $\mathsf{po} \,;\, \mathsf{po}^{-1} \subseteq (\mathsf{po} \cup \mathsf{po}^{-1})^?$
(5) Immediate conflict can only appear between relaxed reads: $\mathtt{cf}|_{\mathsf{imm}} \subseteq \mathsf{R}_{\mathsf{ex}} \times \mathsf{R}_{\mathsf{ex}}$
(6) Every write-exclusive has a corresponding read part:
$$\mathsf{W}_{\mathsf{ex}} \subseteq \mathit{rng}([\mathsf{R}_{\mathsf{ex}}] \,;\, (\mathsf{po}|_{\mathit{imm}} \cap =_{\mathtt{loc}}))$$
(7) jf relates write/read pairs of accesses to same location with the same value.
$$\mathtt{jf} \subseteq [\mathsf{W}] \,;\, (\leq_{\mathtt{loc}} \cap =_{\mathtt{val}}) \,;\, [\mathsf{R}]$$
(8) $\mathtt{jf}^{-1}$ is functional: $\mathtt{jf}; \mathtt{jf}^{-1} \subseteq [\mathsf{W}]$
(9) Every read event has a justification event: $\mathsf{R} \subseteq \mathit{rng}(\mathtt{jf})$
(10) ew is a symmetric relation.
(11) ew relates conflicting write events with the same location and value.
$$\mathsf{ew} \subseteq [\mathsf{W}^{\mathsf{rlx}}] \,;\, (\mathtt{cf} \cap =_{\mathtt{loc}} \cap =_{\mathtt{val}}) \,;\, [\mathsf{W}^{\mathsf{rlx}}]$$
(12) co is strict partial order.
(13) co orders write events with the same location
(the initialization event is placed before other events).
$$[\mathsf{W}] \,;\, <_{\mathtt{loc}} \,;\, [\mathsf{W}] \subseteq \mathsf{co} \subseteq [\mathsf{W}] \,;\, \leq_{\mathtt{loc}} \,;\, [\mathsf{W}]$$
(14) co does not link the writes from the same ew^* class:
$$\mathsf{co} \cap \mathsf{ew}^* \subseteq \emptyset$$
(15) co is closed w.r.t. ew^* equivalence classes:
$$\mathsf{ew}^* \,;\, \mathsf{co} \,;\, \mathsf{ew}^* \subseteq \mathsf{co}$$
(16) co is total on writes to the same location w.r.t. ew^* as equivalence relation:
$$\forall x \in \mathsf{Loc}.\ \mathsf{W}_x \times \mathsf{W}_x \subseteq \mathsf{ew}^* \cup \mathsf{co} \cup \mathsf{co}^{-1}$$

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

$$\mathtt{rf} \triangleq (\mathsf{ew}^* \,;\, \mathtt{jf}) \setminus \mathtt{cf} \qquad \text{(reads-from)}$$
$$\mathtt{jfi} \triangleq \mathtt{jf} \cap \mathsf{po} \text{ and } \mathtt{rfi} \triangleq \mathtt{rf} \cap \mathsf{po} \qquad \text{(internal jf/rf)}$$
$$\mathtt{jfe} \triangleq \mathtt{jf} \setminus \mathsf{po} \text{ and } \mathtt{rfe} \triangleq \mathtt{rf} \setminus \mathsf{po} \qquad \text{(external jf/rf)}$$
$$\mathtt{jo} \triangleq \mathtt{jfe} \,;\, (\mathsf{po} \cup \mathtt{jf})^* \qquad \text{(justification order)}$$
$$\mathtt{lbpat} \triangleq \mathtt{cf}_{\mathtt{imm}} \,;\, [\mathit{rng}(\mathtt{jf} \cap (\mathtt{jf}^? \,;\, \mathtt{hb}))] \,;\, \mathsf{po} \qquad \text{(load buffering pattern)}$$

$$\texttt{rmw} \triangleq [\mathsf{R}_{\mathsf{ex}}] ; \mathsf{po}|_{imm} ; [\mathsf{W}_{\mathsf{ex}}] \qquad \text{(read-modify-write pairs)}$$

$$\texttt{sw} \triangleq [\mathsf{E}^{\sqsupseteq \mathtt{rel}}] ; ([\mathsf{F}] ; \mathsf{po})^? ; (\texttt{jf} ; \texttt{rmw})^* ; \texttt{jf} ; (\mathsf{po} ; [\mathsf{F}])^? ; [\mathsf{E}^{\sqsupseteq \mathtt{acq}}] \qquad \text{(synchronizes-with)}$$

$$\texttt{hb} \triangleq (\mathsf{po} \cup \texttt{sw})^+ \qquad \text{(happens before)}$$

$$\texttt{ecf} \triangleq (\texttt{hb}^{-1})^? ; \texttt{cf} ; \texttt{hb}^? \qquad \text{(extended conflict)}$$

$$\texttt{fr} \triangleq S.\texttt{rf}^{-1} ; S.\texttt{co} \qquad \text{(from-reads)}$$

$$\texttt{eco} \triangleq (\texttt{co} \cup \texttt{rf} \cup \texttt{fr})^+ \qquad \text{(extended coherence order)}$$

$$\texttt{scb} \triangleq \mathsf{po} \cup \mathsf{po}|_{\neq \mathtt{loc}} ; \texttt{hb} ; \mathsf{po}|_{\neq \mathtt{loc}} \cup \texttt{hb}|_{=\mathtt{loc}} \cup \texttt{co} \cup \texttt{fr} \qquad \text{(SC-before)}$$

$$\texttt{psc}_{\text{base}} \triangleq ([\mathsf{E}^{\mathtt{sc}}] \cup [\mathsf{F}^{\mathtt{sc}}] ; \texttt{hb}^?) ; \texttt{scb} ; ([\mathsf{E}^{\mathtt{sc}}] \cup \texttt{hb}^? ; [\mathsf{F}^{\mathtt{sc}}])$$

$$\texttt{psc}_{\mathsf{F}} \triangleq [\mathsf{F}^{\mathtt{sc}}]; (\texttt{hb} \cup \texttt{hb}; \texttt{eco}; \texttt{hb}); [\mathsf{F}^{\mathtt{sc}}]$$

$$\texttt{psc} \triangleq \texttt{psc}_{\text{base}} \cup \texttt{psc}_{\mathsf{F}} \qquad \text{(partial SC order)}$$

For event $e \in S.\mathsf{E}$, we call the set $dom(\texttt{jo} ; [e])$ 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,

$$C \subseteq S.\mathsf{E} \text{ and } [C] ; \texttt{cf} ; [C] = \emptyset \text{ and } dom(\mathsf{po} ; [C]) \subseteq C \text{ and } C \cap \mathsf{R} \subseteq rng([C] ; \texttt{rf})$$

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, $G.\texttt{rf} = S.\texttt{rf}|_C$).

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

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

- $G.\texttt{rmw} \cap (G.\texttt{fr} ; G.\texttt{co}) = \emptyset$. (RMW-ATOMIC)
- $G.\texttt{psc}$ 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 $\text{LBRF}_{\text{RC11}}$ 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:

- $\mathsf{po} \cup \texttt{rf}$ is acyclic (RF-NO-THIN-AIR)
- $\texttt{hb} ; \texttt{eco}^?$ 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 $S.\mathtt{jf} = G.\mathtt{rf}$ and $G.\mathtt{cf} = \emptyset$). Clearly, $S \triangleright G$. Therefore, we only need to show that S is a Weakestmo2-consistent event structure.

Since $S.\mathtt{cf} = \emptyset$, axioms non-contradictory, well-justified are trivially satisfied. By well-formedness, we also have that $S.\mathtt{ew} = \emptyset$ 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, $(S.\mathsf{po} \cup S.\mathtt{jf}) = (G.\mathsf{po} \cup G.\mathtt{rf})$ and thus its acyclicity follows directly from rf-no-thin-air. □

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.

$$\mathtt{rpo} \triangleq [\mathtt{R}^{\mathtt{rlx}}] ; (\mathsf{po} \setminus (\mathsf{po} ; [\mathtt{F}] ; \mathsf{po})) ; [\mathtt{W}^{\mathtt{rlx}}]$$

Definition D.4. 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:

- $G.\mathtt{loc}(e_1) = G.\mathtt{loc}(e_2)$
- $\langle e_1, e_2 \rangle \notin G.\mathtt{hb} \cup G.\mathtt{hb}^{-1}$
- $\langle e_1, e_2 \rangle \in [\mathtt{R}^{\mathtt{rlx}}] ; \mathtt{rpo} ; \mathtt{rfe} ; (\mathsf{po} \cup \mathtt{rf})^* ; [\mathtt{W}]$

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 $\forall e \in S.\mathsf{E}$ we have $\lfloor e \rfloor_{(\mathsf{po} \cup \mathtt{jf})^*} \subseteq S.\mathsf{E}$ where $\lfloor e \rfloor_r \triangleq dom(r ; [e])$

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

Lemma D.8. *For any well-formed S and any $E \subseteq S.\mathsf{E}$ 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. □

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 $E \subseteq S.\mathsf{E}$ if $S|_E$ is prefix-closed then $S|_E$ is promise-free-consistent as well.*

Proof. Follows directly from the fact that properties non-contradictory, 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 □

Definition D.11. Event structure S is *promise-free* if it satisfies the following constraint.

- $\mathtt{cf} \cap \mathtt{jo} = \emptyset$ (promise-free)

Lemma D.12. *If S is* promise-free *then* $\mathtt{cf} \cap (\mathsf{po} \cup \mathtt{jf})^+ = \emptyset$

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

$$(r_1 \cup r_2)^+ = r_1^+ \cup r_1^* ; (r_2 ; r_1^*)^*$$

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

$$\mathtt{cf} \cap (\mathsf{po} \cup \mathtt{jf})^+ = \mathtt{cf} \cap (\mathsf{po} \cup \mathtt{jf} \setminus \mathsf{po})^+ =$$
$$= \mathtt{cf} \cap (\mathsf{po} \cup \mathsf{po}^? ; ((\mathtt{jf} \setminus \mathsf{po}) ; \mathsf{po}^?)^*)$$

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

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

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

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

Proof. Follows trivially from the definition of $\mathtt{rf}$, lemma D.13 and well-justified axiom.

$$\mathtt{rf} \triangleq (\mathsf{ew}^* ; \mathtt{jf}) \setminus \mathtt{cf} = \mathtt{jf} \setminus \mathtt{cf} = \mathtt{jf}$$

□

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 $S.\mathsf{po} \cup S.\mathtt{rf} = S.\mathsf{po} \cup S.\mathtt{jf}$. From axiom no-thin-air we know that $S.\mathsf{po} \cup S.\mathtt{jf}$ is acyclic and therefore $G.\mathsf{po} \cup G.\mathtt{rf} \subseteq S.\mathsf{po} \cup S.\mathtt{rf} = S.\mathsf{po} \cup S.\mathtt{jf}$ is acyclic as well.
- coherent
  Follows from the coherence of S and the fact that $G.\mathtt{hb} ; G.\mathsf{eco}^? \subseteq S.\mathtt{hb} ; S.\mathsf{eco}^?$.

□

Lemma D.16. *Given consistent event structure S, if additionally it is* promise-free *then for every event $e \in S.\mathsf{E}$ its $(\mathsf{po} \cup \mathtt{jf})$ prefix is extractable execution graph, i.e., for $G \triangleq S|_E$, where $E = \lfloor e \rfloor_{(\mathsf{po} \cup \mathtt{jf})^*}$, 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 $(\mathsf{po} \cup \mathtt{jf})^*$ downward closure of the event e, then clearly it is po downward-closed. $\mathtt{rf}$ completeness follows from the fact that $\mathtt{jf} = \mathtt{rf}$ by lemma D.14, $\mathtt{jf}$ is complete by well-formedness of S, and E is $(\mathsf{po} \cup \mathtt{jf})^*$ 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 $(\mathsf{po} \cup \mathtt{jf})$ 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*

$$\mathtt{cf} \cap \mathtt{jo} \subseteq \mathsf{rpo}^{-1} ; \mathtt{cf}|_{imm} ; \mathsf{po}^?$$

Proof. Let $\langle w, e\rangle \in \mathtt{cf} \cap \mathtt{jo}$. Then there should exists $\langle r, r'\rangle \in \mathtt{cf}|_{imm}$, s.t. $\langle r, w\rangle \in \mathsf{po}^?$ and $\langle r', e\rangle \in \mathsf{po}^?$. By well-formedness $r, r' \in S.\mathtt{R}^{\mathtt{rlx}}$, and $w \in S.\mathtt{W}$. Therefore $\langle r, w\rangle \in \mathsf{po}$. It is left to show that $\langle r, w\rangle \in \mathtt{rpo}$. By certified there should exists w', s.t. $\langle w, w'\rangle \in \mathtt{ew}$ and $\langle w', e\rangle \in (\mathsf{po} \cup \mathsf{po}^{-1})^?$. Then we get that $w, w' \in \mathtt{W}^{\mathtt{rlx}}$ 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 $\langle f, w\rangle \in [F]\,;\mathsf{po}$. But since $\langle r, f\rangle \in \mathsf{po}$ we have $\langle f, w'\rangle \in \mathtt{cf}$. Therefore $\langle f, w'\rangle \notin \mathsf{po}$ which contradicts well-fenced. □

Lemma D.19. *In the consistent event structure S promises can only be relaxed:*

$$dom(\mathtt{cf} \cap \mathtt{jo}) \subseteq \mathtt{W}^{\mathtt{rlx}}$$

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

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

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 $\langle r, p\rangle \in \mathsf{po}$ and $\langle r, r'\rangle \in \mathtt{cf}_{\mathtt{imm}}$. Therefore, since conflict relation propogates along program order, we have that $\langle p, r'\rangle \in \mathtt{cf}$. It is left to show that $\langle p, r'\rangle \in \mathtt{jo}$.

Suppose that $\langle p, r'\rangle \notin \mathtt{jo}$. Let r'' be the first event in a branch between r' and e that observes p:

- $\langle r', r''\rangle \in \mathsf{po}$
- $\langle r'', e\rangle \in \mathsf{po}$
- $\langle p, r''\rangle \in \mathtt{jo}$
- $\langle p, r''\rangle \notin \mathtt{jo}\,;\mathsf{po}$

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

Suppose $\langle w'', r''\rangle \in \mathtt{jf}^?\,;\mathtt{hb}$. Note that $\mathtt{jf}^?\,;\mathtt{hb} \subseteq (\mathsf{po} \cup \mathtt{jf})^+$. If the last edge in the $\mathtt{jf}^?\,;\mathtt{hb}$ path from w'' to r'' is po edge then we would have $\langle p, r''\rangle \in \mathtt{jo}\,;(\mathsf{po} \cup \mathtt{jf})^+\,;\mathsf{po} \subseteq \mathtt{jo}\,;\mathsf{po}$. This contradicts our assumption. Therefore the last edge in the $\mathtt{jf}^?\,;\mathtt{hb}$ path has to be $\mathtt{jfe}\,;[E^{\sqsupseteq\mathtt{acq}}]$. But then, by well-fenced it should be the case that $\langle w'', p\rangle \in \mathtt{jfe}\,;[E^{\sqsupseteq\mathtt{acq}}]\,;\mathsf{po} \subseteq (\mathsf{po} \cup \mathtt{jf})^*$ and since also $\langle p, w''\rangle \in \mathtt{jo} \subseteq (\mathsf{po} \cup \mathtt{jf})^*$ we have a $(\mathsf{po} \cup \mathtt{jf})$ cycle, which contradicts axiom no-thin-air. Therefore $\langle w'', r''\rangle \notin \mathtt{jf}^?\,;\mathtt{hb}$. In other words r'' is justified externally. Thus we have $\langle w'', p\rangle \in (\mathtt{jf} \setminus \mathtt{jf}^?\,;\mathtt{hb})\,;\mathsf{po}\,;\mathtt{ew}$

Also note that $\langle r'', p\rangle \notin \mathtt{lbpat}$, since $\langle r'', p\rangle \notin \mathtt{cf}_{\mathtt{imm}}\,;\mathsf{po}$ (the later is true since we know $\langle r', p\rangle \in \mathtt{cf}_{\mathtt{imm}}\,;\mathsf{po}$ and $\langle r', r''\rangle \in \mathsf{po}$).

Thereby the preconditions of axiom no-bait-and-switch apply to $\langle w'', p\rangle$, and we can conclude that $\langle w'', p\rangle \in \mathtt{jf}\,;\mathsf{po} \subseteq (\mathsf{po} \cup \mathtt{jf})^*$. But then again, using the fact that $\langle p, w''\rangle \in \mathtt{jo} \subseteq (\mathsf{po} \cup \mathtt{jf})^*$, we arrive at contradiction, because we have a $(\mathsf{po} \cup \mathtt{jf})^*$ cycle forbidden by axiom no-thin-air. □

Lemma D.21. *Let P be LB-race-free under* RC11. *Then every consistent event structure S of P is* promise-free.

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

- $\forall i \in \{1, \dots, n\}.\ e_i \in S.\mathsf{E}$
- $\forall e \in S.\mathsf{E}.\ \exists i.\ e = e_i$
- $\forall i\ j \in \{1, \dots, n\}.\ i < j \Rightarrow \langle e_i, e_j \rangle \in R$

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

Base Case: $P(i = 0)$.
$S_1.\mathsf{E} = \{e_0\}$. In this case $S_1.\mathtt{jo} = \emptyset$. Thus PROMISE-FREE is trivially satisfied.

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

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

$$\begin{aligned}
\mathsf{po}_{i+1} &= \mathsf{po}_i \cup \mathsf{po}_{i+1} ; [e_{i+1}] \\
\mathtt{jf}_{i+1} &= \mathtt{jf}_i \cup \mathtt{jf}_{i+1} ; [e_{i+1}] \\
\mathtt{jo}_{i+1} &= \mathtt{jo}_i ; ((\mathsf{po}_{i+1} \cup \mathtt{jf}_{i+1}) ; [e_{i+1}])^? \cup \\
&\quad \cup \mathtt{jf}_{i+1} ; [e_{i+1}] \setminus \mathsf{po}_{i+1}
\end{aligned}$$

Using these equalities we prove PROMISE-FREE by case analysis on e_{i+1}.

- $e_{i+1} \in S.\mathsf{F}$
  In this case $\mathtt{jf}_{i+1} ; [e_{i+1}] = \emptyset$ and thus $\mathtt{jo}_{i+1} = \mathtt{jo}_i ; (\mathsf{po}_{i+1} ; [e_{i+1}])^?$. If e_{i+1} has no po_{i+1}-predecessors, then $\mathtt{jo}_{i+1} = \mathtt{jo}_i$ and thus we have $\mathtt{cf}_{i+1} \cap \mathtt{jo}_{i+1} = \mathtt{cf}_i \cap \mathtt{jo}_i = \emptyset$. Otherwise we have

  $$\begin{aligned}
  \mathtt{cf}_{i+1} \cap \mathtt{jo}_{i+1} = \mathtt{cf}_i \cap \mathtt{jo}_i \cup \\
  \cup \mathtt{cf}_{i+1} \cap (\mathtt{jo}_i ; \mathsf{po}_{i+1} ; [e_{i+1}])
  \end{aligned}$$

  We have $\mathtt{cf}_i \cap \mathtt{jo}_i = \emptyset$ by inductive assumption. We prove $\mathtt{cf}_{i+1} \cap (\mathtt{jo}_i ; \mathsf{po}_{i+1} ; [e_{i+1}]) = \emptyset$ by contradiction. Let $p \in dom(\mathtt{cf}_{i+1} \cap (\mathtt{jo}_i ; \mathsf{po}_{i+1} ; [e_{i+1}]))$. Note that $p \in S.\mathsf{E}_i$. Let e_j be immediate po_{i+1}-predecessor of e_{i+1}. Then $p \in dom(\mathtt{jo}_{i+1} ; \mathsf{po}_{i+1} ; [e_{i+1}])$ implies $p \in dom(\mathtt{jo}_i ; [e_j])$. Event e_j cannot be in conflict with p, since it would violate inductive assumption $\mathtt{cf}_i \cap \mathtt{jo}_i = \emptyset$. Thus either $\langle p, e_j \rangle \in \mathsf{po}_i$ or $\langle e_j, p \rangle \in \mathsf{po}_i$. If $\langle p, e_j \rangle \in \mathsf{po}_i$ then $\langle p, e_{i+1} \rangle \in \mathsf{po}_{i+1}$, but it contradicts $\langle p, e_{i+1} \rangle \in \mathtt{cf}_{i+1}$. If $\langle e_j, p \rangle \in \mathsf{po}_i$, then since $\langle p, e_j \rangle \in \mathtt{jo}_i$ by $\mathtt{jo} ; \mathsf{po} \subseteq \mathtt{jo}$, which follows immediately from the definition of $\mathtt{jo}$, we have $\langle p, p \rangle \in \mathtt{jo}_i$. But $\mathtt{jo}_i \subseteq (\mathsf{po}_i \cup \mathtt{jf}_i)^+$ and thus we have contradiction with NO-THIN-AIR. Therefore $\mathtt{cf}_{i+1} \cap \mathtt{jo}_{i+1} = \emptyset$.
- $e_{i+1} \in S.\mathsf{W}$
  This case is similar to the previous one.
- $e_{i+1} \in S.\mathsf{R}$
  In this case we have:

  $$\begin{aligned}
  \mathtt{cf}_{i+1} \cap \mathtt{jo}_{i+1} = \mathtt{cf}_i \cap \mathtt{jo}_i \cup \\
  \cup \mathtt{cf}_{i+1} \cap (\mathtt{jf}_{i+1} ; [e_{i+1}] \setminus \mathsf{po}_{i+1}) \cup \\
  \cup \mathtt{cf}_{i+1} \cap (\mathtt{jo}_i ; \mathsf{po}_{i+1} ; [e_{i+1}]) \cup \\
  \cup \mathtt{cf}_{i+1} \cap (\mathtt{jo}_i ; \mathtt{jf}_{i+1} ; [e_{i+1}])
  \end{aligned}$$

  By inductive assumption $\mathtt{cf}_i \cap \mathtt{jo}_i = \emptyset$. By well-formedness $\mathtt{cf}_{i+1} \cap \mathtt{jf}_{i+1} = \emptyset$ and thus $\mathtt{cf}_{i+1} \cap (\mathtt{jf}_{i+1} ; [e_{i+1}] \setminus \mathsf{po}_{i+1}) = \emptyset$. $\mathtt{cf}_{i+1} \cap (\mathtt{jo}_i ; \mathsf{po}_{i+1} ; [e_{i+1}]) = \emptyset$ by the same argument as in cases $e_{i+1} \in S.\mathsf{F}$ and $e_{i+1} \in S.\mathsf{W}$.

We only left to show that $\mathtt{cf}_{i+1} \cap (\mathtt{jo}_i ; \mathtt{jf}_{i+1} ; [e_{i+1}]) = \emptyset$. Let e_j be immediate $\mathtt{po}_{i+1}$-predecessor of e_{i+1} and let $p \in dom(\mathtt{cf}_{i+1} \cap (\mathtt{jo}_i ; \mathtt{jf}_{i+1} ; [e_{i+1}]))$. By lemma D.20 the branch of e_{i+1} should observe promise p up to the point of immediate conflict. Therefore if $\langle p, e_j \rangle \in \mathtt{cf}_i$ it should also be true that $\langle p, e_j \rangle \in \mathtt{jo}_i$. But it would violate the inductive assumption $\mathtt{cf}_i \cap \mathtt{jo}_i = \emptyset$. Thus $\langle p, e_j \rangle \notin \mathtt{cf}_i$.

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

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

Suppose $\langle r, w \rangle \in \mathtt{hb}_i$. Since $\langle r, w \rangle \notin \mathtt{po}_i$, by definition of $\mathtt{hb}_i$, there should exists either.

- Write $w_{\mathtt{rel}} \in S.\mathtt{W}^{\sqsupseteq \mathtt{rel}}$, s.t. $\langle r, w_{\mathtt{rel}} \rangle \in S.\mathtt{po}_i$ and $\langle w_{\mathtt{rel}}, w \rangle \in \mathtt{hb}_i \setminus \mathtt{po}_i$. But in this case we have $\langle w_{\mathtt{rel}}, e_{i+1} \rangle \in \mathtt{cf}_{i+1} \cap \mathtt{jo}_{i+1}$, i.e., $w_{\mathtt{rel}}$ is a promise. Then by lemma D.19 $w_{\mathtt{rel}} \in S.\mathtt{W}^{\mathtt{rlx}}$ which contradicts the fact that $w_{\mathtt{rel}}$ is a release write.
- Fence $f_{\mathtt{rel}} \in S.\mathtt{F}^{\sqsupseteq \mathtt{rel}}$, s.t. $\langle r, f_{rel} \rangle \in \mathtt{po}_i$ and $\langle f_{\mathtt{rel}}, w \rangle \in \mathtt{hb}_i$. But in this case $\langle f_{\mathtt{rel}}, p \rangle \in \mathtt{po}_i$ which contradicts the fact that $\langle r, p \rangle \in \mathtt{rpo}_i$.

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

□

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 PROMISE-FREE. Therefore, by lemma D.15, G is RC11 consistent execution graph of P.

□

E EXTENDING THE LBRF THEOREM TO THE FULL WEAKESTMO2 MODEL

In this section, we describe how the proof of the $\mathsf{LBRF}_{\mathsf{RC11}}$ 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:

- $\mathtt{po} \cup \mathtt{rf}$ is acyclic (RF-NO-THIN-AIR)
- $\mathtt{hb} ; \mathtt{eco}^?$ is irreflexive. (COHERENT)
- $\mathtt{rmw} \cap (\mathtt{fr} ; \mathtt{co}) = \emptyset$. (RMW-ATOMIC)

- $\mathtt{psc}$ is acyclic. (SEQUENTIAL-CONSISTENCY)

Next, note that in the proof of D.21 at some point we have to construct RC11 consistent execution graph from the $\mathsf{po} \cup \mathtt{jf}$ prefix of the event structure. There, the fact that $\mathsf{po} \cup \mathtt{jf}$ 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 $\mathsf{po} \cup \mathtt{jf}$ 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:

$$a : \mathsf{R}^{rlx}_{\mathsf{ex}} \,;\, b : \mathsf{W}^{rlx}_{\mathsf{ex}} \,;\, c : \mathsf{W}^{rlx} \rightsquigarrow c : \mathsf{W}^{rlx} \,;\, a : \mathsf{R}^{rlx}_{\mathsf{ex}} \,;\, b : \mathsf{W}^{rlx}_{\mathsf{ex}}$$

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 $\mathsf{po} \cup \mathtt{jf}$ conflict-free prefix of the event structure would satisfy RMW-ATOMIC. We managed to do that with the assumption that P has no $\mathtt{sc}$ 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:

- $G.\mathtt{loc}(e_1) = G.\mathtt{loc}(e_2)$
- $\langle e_1, e_2\rangle \notin G.\mathtt{hb} \cup G.\mathtt{hb}^{-1}$
- Either
  - $\langle e_1, e_2\rangle \in [\mathsf{R}^{\mathtt{rlx}}] \,;\, \mathtt{rpo} \,;\, (\mathtt{rf} \setminus \mathsf{po}) \,;\, (\mathsf{po} \cup \mathtt{rf})^* \,;\, [\mathtt{W}]$ or
  - $\langle e_1, e_2\rangle \in [\mathsf{W}^{\mathtt{rlx}} \cup \mathsf{R}^{\mathtt{rlx}}_{\mathsf{ex}}] \,;\, (\mathsf{po} \cup \mathtt{rf})^+ \setminus \mathtt{rf} \,;\, [\mathsf{R}^{\mathtt{rlx}}_{\mathsf{ex}}]$

Definition E.3. Event structure S is *prefix-rmw-atomic* if for every event $e \in S.\mathsf{E}$ its $(\mathsf{po} \cup \mathtt{jf})$ prefix satisfies atomicity, i.e., for $E = \lfloor e \rfloor_{(\mathsf{po} \cup \mathtt{jf})^*}$ the following is true:

- $\mathtt{rmw}|_E \cap (\mathtt{fr}|_E \,;\, \mathtt{co}|_E) = \emptyset$ (PREFIX-RMW-ATOMIC)

Definition E.4. A triple of events $\langle w, r_{\mathsf{ex}}, w_{\mathsf{ex}}\rangle$ is an atomicity violation in an execution graph G if

- $\langle w_{\mathsf{ex}}, r_{\mathsf{ex}}\rangle \in G.\mathtt{rmw}$,
- $\langle r_{\mathsf{ex}}, w\rangle \in G.\mathtt{fr}$, and
- $\langle w, w_{\mathsf{ex}}\rangle \in G.\mathtt{co}$.

Definition E.5. An atomicity violation $\langle w, r_{\mathsf{ex}}, w_{\mathsf{ex}}\rangle$ in execution graph G is $\mathtt{co}$*-repairable* if the following conditions are met:

- $w \notin G.\mathsf{W}_{\mathsf{ex}}$.
- Either w or w_{ex} is $(\mathsf{po} \cup \mathtt{rf})^+$ maximal.

Definition E.6. Two executions are equivalent modulo $\mathtt{co}$, written $G \cong_{\mathtt{co}} G'$, if they agree on all components except $\mathtt{co}$, that is:

(1) $G'.\mathsf{E} = G.\mathsf{E}$.
(2) $G'.\mathtt{tid} = G.\mathtt{tid}$
(3) $G'.\mathtt{lab} = G.\mathtt{lab}$
(4) $G'.\mathsf{po} = G.\mathsf{po}$.
(5) $G'.\mathtt{rf} = G.\mathtt{rf}$.

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, $\langle w, r_{\mathsf{ex}}, w_{\mathsf{ex}}\rangle$, and further suppose*

that the violation is co*-repairable. Then there exists* RC11*-consistent execution graph* G' *equivalent to* G *modulo* co.

Proof. Let $G' \triangleq \langle G.\mathsf{E}, G.\mathtt{tid}, G.\mathtt{lab}, G.\mathsf{po}, G.\mathtt{rf}, \mathtt{id}, \mathtt{co}' \rangle$ where

$$\Delta_{\mathtt{co}} = \{\langle w, w_{\mathsf{ex}} \rangle\} \quad \text{and} \quad \mathtt{co}' = G.\mathtt{co} \setminus \Delta_{\mathtt{co}} \cup \Delta_{\mathtt{co}}^{-1}$$

That is, co′ is the same as co except that we swapped a single edge from $\langle w, w_{\mathsf{ex}} \rangle$ to $\langle w_{\mathsf{ex}}, w \rangle$. Note that G' trivially satisfies condition $G' \cong_{\mathtt{co}} G$.

Let us show that 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 co.

Consider the following inequality:

$$\begin{aligned}
&(G.\mathtt{co} \setminus \Delta_{\mathtt{co}} \cup \Delta_{\mathtt{co}}^{-1}) \,;\, (G.\mathtt{co} \setminus \Delta_{\mathtt{co}} \cup \Delta_{\mathtt{co}}^{-1}) = \\
&= (G.\mathtt{co} \setminus \Delta_{\mathtt{co}}) \,;\, (G.\mathtt{co} \setminus \Delta_{\mathtt{co}}) \cup \\
&\cup (G.\mathtt{co} \setminus \Delta_{\mathtt{co}}) \,;\, \Delta_{\mathtt{co}}^{-1} \cup \\
&\cup \Delta_{\mathtt{co}} \,;\, (G.\mathtt{co} \setminus \Delta_{\mathtt{co}}^{-1}) \overset{?}{\subseteq} \\
&\overset{?}{\subseteq} G.\mathtt{co} \setminus \Delta_{\mathtt{co}} \cup \Delta_{\mathtt{co}}^{-1}
\end{aligned}$$

First note that $(G.\mathtt{co} \setminus \Delta_{\mathtt{co}}) \,;\, (G.\mathtt{co} \setminus \Delta_{\mathtt{co}}) \subseteq (G.\mathtt{co} \setminus \Delta_{\mathtt{co}})$. Indeed, suppose there exists w', s.t. $\langle w, w' \rangle \in G.\mathtt{co}$ and $\langle w', w_{\mathsf{ex}} \rangle \in G.\mathtt{co}$. But then, since $\langle w_{\mathsf{ex}}, w \rangle \in \mathtt{fr}$ we would also have $\langle w_{\mathsf{ex}}, w' \rangle \in \mathtt{fr}$. Therefore $\langle w', r_{\mathsf{ex}}, w_{\mathsf{ex}} \rangle$ would form an atomicity violation in G, which contradicts the assumption that $\langle w, r_{\mathsf{ex}}, w_{\mathsf{ex}} \rangle$ is single atomicity violation. For the same reasons $(G.\mathtt{co} \setminus \Delta_{\mathtt{co}}) \,;\, \Delta_{\mathtt{co}}^{-1} \subseteq G.\mathtt{co} \setminus \Delta_{\mathtt{co}}$ and $\Delta_{\mathtt{co}} \,;\, (G.\mathtt{co} \setminus \Delta_{\mathtt{co}}^{-1}) \subseteq G.\mathtt{co} \setminus \Delta_{\mathtt{co}}$. Therefore we have:

$$\mathtt{co}' \triangleq (G.\mathtt{co} \setminus \Delta_{\mathtt{co}} \cup \Delta_{\mathtt{co}}^{-1})^{+} = G.\mathtt{co} \setminus \Delta_{\mathtt{co}} \cup \Delta_{\mathtt{co}}^{-1}$$

Note that irreflexivity of relation r is equivalent to $r \cap \mathtt{id} = \emptyset$ and thus irreflexivity of the union of relations is equivalent to irreflexivity of components. Irreflexivity $\mathtt{co} \setminus \Delta_{\mathtt{co}}$ follows from irreflexivity of co, and $\Delta_{\mathtt{co}}^{-1}$ is trivially irreflexive. Therefore co′ is well-formed and thus G' is well-formed as well.

Since $G'.\mathsf{po} = G.\mathsf{po}$, $G'.\mathtt{rf} = G.\mathtt{rf}$, and G satisfies RF-NO-THIN-AIR we have that G' also satisfies RF-NO-THIN-AIR.

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

Also note that since hb is defined in terms of lab, po, and rf we have that $G'.\mathtt{hb} = G.\mathtt{hb}$. Therefore $G.\mathtt{hb}'$ is irreflexive.

We need to show $G'.\mathtt{hb} \,;\, G'.\mathtt{eco}^{?}$ is irreflexive.

First, note that for any well-formed execution graph (or, more general, event structure) the following equation can be proven:

$$\begin{aligned}
\mathtt{eco} &\triangleq (\mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr})^{+} = \\
&= \mathtt{rf} \cup \mathtt{co} \,;\, \mathtt{rf}^{?} \cup \mathtt{fr} \,;\, \mathtt{rf}^{?}
\end{aligned}$$

Then observe that:

$$G'.\mathtt{eco} = G.\mathtt{rf} \cup ;G'.\mathtt{co} \,;\, G.\mathtt{rf}^{?} \cup G.\mathtt{rf}^{-1} \,;\, G'.\mathtt{co} \,;\, G.\mathtt{rf}^{?}$$

Irreflexivity of the following components follows from the fact that G is COHERENT.

- $G.\mathtt{hb} \,;\, G.\mathtt{rf}$ is irreflexive.

- $G.\mathsf{hb}\,;((G.\mathsf{co}\setminus\Delta_{\mathsf{co}})\,;G.\mathtt{rf}^?)$ is irreflexive.
- $G.\mathsf{hb}\,;(G.\mathtt{rf}^{-1}\,;(G.\mathsf{co}\setminus\Delta_{\mathsf{co}})\,;G.\mathtt{rf}^?)$ is irreflexive.

We only left to show irreflexivity of the following components

- $G.\mathsf{hb}\,;\Delta_{\mathsf{co}}^{-1}\,;\mathtt{rf}^?$.
- $G.\mathsf{hb}\,;\mathtt{rf}^{-1}\,;\Delta_{\mathsf{co}}^{-1}\,;\mathtt{rf}^?$.

Note that $\forall e\in G.\mathsf{E}.\ \langle e,w_{\mathsf{ex}}\rangle\in\mathsf{hb}\Leftrightarrow\langle e,r_{\mathsf{ex}}\rangle\in\mathsf{hb}$. Suppose that $\langle w,w_{\mathsf{ex}}\rangle\in G.\mathtt{rf}^?\,;G.\mathsf{hb}$. Then it would imply $\langle w,r_{\mathsf{ex}}\rangle\in G.\mathtt{rf}^?\,;G.\mathsf{hb}$ and therefore we would have a cycle $\langle r_{\mathsf{ex}},r_{\mathsf{ex}}\rangle\in G.\mathtt{fr}\,;G.\mathtt{rf}^?\,;G.\mathsf{hb}$ which contradicts the assumption that G is COHERENT.

Next, suppose that $\langle w,w_{\mathsf{ex}}\rangle\in G.\mathtt{rf}^?\,;G.\mathsf{hb}\,;G.\mathtt{rf}^{-1}$. By our assumption either w or w_{ex} is $(G.\mathsf{po}\cup G.\mathtt{rf})^+$ maximal. If w_{ex} is $(G.\mathsf{po}\cup G.\mathtt{rf})^+$ maximal then $G.\mathtt{rf}^{-1}\,;[w_{\mathsf{ex}}]$ is empty. Similarly, $[w]\,;G.\mathtt{rf}$ is empty if w is maximal. Thus we have only left to consider the case $\langle w,w_{\mathsf{ex}}\rangle\in G.\mathsf{hb}\,;G.\mathtt{rf}^{-1}$ and w is $(G.\mathsf{po}\cup G.\mathtt{rf})^+$ maximal. But then notice that $G.\mathsf{hb}\subseteq(G.\mathsf{po}\cup G.\mathtt{rf})^+$ and thus $[w]\,;G.\mathsf{hb}$ is empty as well. □

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

- $w'_{\mathsf{ex}}\in G.\mathtt{W}_{\mathsf{ex}}$ and
- w_{ex} is $\mathsf{po}\cup\mathtt{rf}$ maximal.

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

(1) $G'.\mathsf{E}=G.\mathsf{E}\setminus\{w_{\mathsf{ex}},r_{\mathsf{ex}}\}\cup\{r'_{\mathsf{ex}}\}$ *where*
- $r'_{\mathsf{ex}}\triangleq\langle G.\mathtt{id}(r_{\mathsf{ex}}),G.\mathtt{tid}(r_{\mathsf{ex}}),lab\rangle$
- $lab\triangleq\mathsf{R}^o_{\mathsf{ex}}(l,v)$
- $o\triangleq G.\mathtt{mod}(r_{\mathsf{ex}})\quad l\triangleq G.\mathtt{loc}(r_{\mathsf{ex}})\quad v\triangleq G.\mathtt{val}(w'_{\mathsf{ex}})$

(2) $G'.\mathsf{po}=G.\mathsf{po}|_{G.\mathsf{E}'}$.
(3) $G'.\mathtt{rf}=G.\mathtt{rf}|_{G.\mathsf{E}'\setminus\{r_{\mathsf{ex}}\}}\cup\Delta_{\mathtt{rf}}$ *where* $\Delta_{\mathtt{rf}}\triangleq\{\langle w'_{\mathsf{ex}},r'_{\mathsf{ex}}\rangle\}$
(4) $G'.\mathsf{co}=G.\mathsf{co}|_{G.\mathsf{E}'}$.

PROOF. It is easy to show that G' is well-formed.

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

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

Thus we only left to show that $G'.\mathsf{hb}\,;G'.\mathsf{eco}^?$ is irreflexive.

- $G'.\mathsf{hb}$ is irreflexive since $G'.\mathsf{hb}\subseteq(G.\mathsf{po}\cup G.\mathtt{rf})^+$.
- Irreflexivity of $G'.\mathsf{hb}\,;G'.\mathtt{rf}$ follows from irreflexivity of $G.\mathsf{hb}\,;G.\mathtt{rf}$ and the fact that r'_{ex} is $(G'.\mathsf{po}\cup G'.\mathtt{rf})$ maximal.
- Irreflexivity of $G'.\mathsf{hb}\,;G'.\mathsf{co}=G'.\mathsf{hb}\,;G.\mathsf{co}=G.\mathsf{hb}\,;G.\mathsf{co}$ follows directly.
- Let us consider $G'.\mathsf{hb}\,;G'.\mathsf{co}\,;G'.\mathtt{rf}^?$.

$$G'.\mathsf{hb}\,;G'.\mathsf{co}\,;G'.\mathtt{rf}^?=G.\mathsf{hb}\,;G.\mathsf{co}\,;G'.\mathtt{rf}=$$
$$G.\mathsf{hb}\,;G.\mathsf{co}\,;(G.\mathtt{rf}\cup\Delta_{\mathtt{rf}})$$

  But the cycle $\langle r'_{\mathsf{ex}},r'_{\mathsf{ex}}\rangle\in G.\mathsf{hb}\,;G.\mathsf{co}\,;\Delta_{\mathtt{rf}}$ would contradict $G'.\mathsf{po}\cup G'.\mathtt{rf}$ maximality of r'_{ex}.
- Finally, suppose there is cycle $\langle r'_{\mathsf{ex}},r'_{\mathsf{ex}}\rangle\in G'.\mathtt{fr}\,;G'.\mathtt{rf}^?\,;G'.\mathsf{hb}$. Let w be the endpoint of the first edge in the cycle, i.e., $\langle r'_{\mathsf{ex}},w\rangle\in G'.\mathtt{fr}$. Then we would have $\langle w'_{\mathsf{ex}},w\rangle\in G.\mathsf{co}$ and therefore $\langle w,r_{\mathsf{ex}}\rangle\in G.\mathtt{fr}\,;G.\mathtt{rf}^?\,;G.\mathsf{hb}$ would also close a cycle in $G.\mathsf{hb}\,;G.\mathsf{eco}$.

□

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 $S_0.\mathtt{rmw} = \emptyset$. Thus we only consider the step of the induction. Note the following holds:

$$\begin{aligned}
&\mathtt{rmw}_{i+1} = \mathtt{rmw}_i \cup \mathtt{rmw}_{i+1} ; [e_{i+1}] \\
&\mathtt{rmw}_{i+1} \cap (\mathtt{fr}_{i+1} ; \mathtt{co}_{i+1}) = \\
&\qquad = \mathtt{rmw}_i \cap (\mathtt{fr}_i ; \mathtt{co}_i) \\
&\qquad \cup \mathtt{rmw}_i \cap (\mathtt{fr}_i ; [e_{i+1}] ; \mathtt{co}_i) \\
&\qquad \cup (\mathtt{rmw}_{i+1} ; [e_{i+1}]) \cap (\mathtt{fr}_i ; \mathtt{co}_{i+1} ; [e_{i+1}])
\end{aligned}$$

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

- $e_{i+1} \in S.\mathsf{F}$
  In this case $\mathtt{fr}_{i+1} ; [e_{i+1}] = \emptyset$, $\mathtt{co}_{i+1} ; [e_{i+1}] = \emptyset$, and $\mathtt{rmw}_{i+1} ; [e_{i+1}] = \emptyset$. Thus $\mathtt{rmw}_{i+1} \cap (\mathtt{fr}_{i+1} ; \mathtt{co}_{i+1}) = \mathtt{rmw}_i \cap (\mathtt{fr}_i ; \mathtt{co}_i) = \emptyset$. Therefore, for every event $e \in S.\mathsf{E}_{i+1}$, $E = \lfloor e \rfloor_{(\mathtt{po} \cup \mathtt{jf})^*}$ the required property $\mathtt{rmw}_{i+1}|_E \cap (\mathtt{fr}_{i+1}|_E ; \mathtt{co}_{i+1}|_E) = \emptyset$ follows immediately.
- $e_{i+1} \in S.\mathsf{R}$
  This case is similar to the previous one.
- $e_{i+1} \in S.\mathsf{W}$
  For every $e \in S.\mathsf{E}_i$ 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 $E = \lfloor e \rfloor_{(\mathtt{po} \cup \mathtt{jf})^*}$. We need to show:

$$\begin{aligned}
&\mathtt{rmw}_{i+1}|_E \cap (\mathtt{fr}_{i+1}|_E ; \mathtt{co}_{i+1}|_E) = \\
&\qquad = \mathtt{rmw}_i|_E \cap (\mathtt{fr}_i|_E ; \mathtt{co}_i|_E) \\
&\qquad \cup \mathtt{rmw}_i|_E \cap (\mathtt{fr}_i|_E ; [e_{i+1}] ; \mathtt{co}_i|_E) \\
&\qquad \cup (\mathtt{rmw}_{i+1}|_E ; [e_{i+1}]) \cap (\mathtt{fr}_i|_E ; \mathtt{co}_{i+1}|_E ; [e_{i+1}])
\end{aligned}$$

  - $\mathtt{rmw}_i|_E \cap (\mathtt{fr}_i|_E ; \mathtt{co}_i|_E) = \emptyset$ by inductive assumption.
  - Let $\langle r_{\mathsf{ex}}, w_{\mathsf{ex}} \rangle \in \mathtt{rmw}_i|_E$, s.t. $\langle r_{\mathsf{ex}}, e_{i+1} \rangle \in \mathtt{fr}_{i+1}$ and $\langle e_{i+1}, w_{\mathsf{ex}} \rangle \in \mathtt{co}_{i+1}$. Note that by well-formedness we have that $r_{\mathsf{ex}} \in S_i.\mathsf{R}_{\mathsf{ex}}$ and we have $\langle r_{\mathsf{ex}}, e_{i+1} \rangle, \langle w_{\mathsf{ex}}, e_{i+1} \rangle \in (\mathtt{po} \cup \mathtt{jf})^+$. Consider the cases for e_{i+1}.
    * $e_{i+1} \in S.\mathsf{W}_{\mathsf{ex}}$.
      Let e_j be immediate $\mathtt{po}$ predecessor if e_{i+1}. Then by well-formedness $\langle e_j, e_{i+1} \rangle \in S_{i+1}.\mathtt{rmw}$. Then $\langle w_{\mathsf{ex}}, e_j, e_{i+1} \rangle$ forms $\mathtt{rf}$-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 RC11 consistent execution graph G, s.t. $G \cong_{\mathtt{rf}(e_j)} S_{i+1}|_E$. We have $\langle r_{\mathsf{ex}}, e_j \rangle \in [\mathsf{R}_{\mathsf{ex}}] ; (G.\mathtt{po} \cup G.\mathtt{rf})^+ ; [\mathsf{R}_{\mathsf{ex}}]$, i.e., the pair $\langle r_{\mathsf{ex}}, e_j \rangle$ forms load-buffering race in RC11-consistent execution G, which contradicts our assumption that P is LB-race-free.
    * $e_{i+1} \notin S.\mathsf{W}_{\mathsf{ex}}$.
      Then $\langle e_{i+1}, r_{\mathsf{ex}}, w_{\mathsf{ex}} \rangle$ forms a $\mathtt{co}$-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. $G \cong_{\mathtt{co}} S_{i+1}|_E$. Since S_{i+1} is PROMISE-FREE, by D.14

we also have that

$$(\mathsf{po}_{i+1} \cup \mathtt{jf}_{i+1})^+ = (\mathsf{po}_{i+1} \cup \mathtt{rf}_{i+1})^+ = (G.\mathsf{po} \cup G.\mathtt{rf})^+$$

Therefore the pair $\langle r_{\mathsf{ex}}, e_{i+1}\rangle \in [\mathsf{R}_{\mathsf{ex}}]\,;\,(G.\mathsf{po} \cup G.\mathtt{rf})^+\,;\,[\mathsf{R}_{\mathsf{ex}}]$. forms a load-buffering race in RC11-consistent execution G, which contradicts our assumption that P is LB-race-free.

– Let $\langle e_j, e_{i+1}\rangle \in \mathtt{rmw}_{i+1}$. Let $w \in E$, s.t. $\langle e_j, w\rangle \in \mathtt{fr}_i$ and $\langle w, e_{i+1}\rangle \in \mathtt{co}_{i+1}$. 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.

□

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 $G \vdash TC \xrightarrow{e} TC'$ where TC and TC' are *traversal configurations* and e is an event being traversed. A traversal configuration is a tuple $\langle C, I\rangle$, where $C \subseteq G.\mathsf{E}$ is a set of *covered events* and $I \subseteq G.\mathtt{W}$ is a set of *issued writes*.

Giving the operational semantics of traversal $G \vdash TC \xrightarrow{e} TC'$, such that $G.\mathtt{tid}(e) = t$, and the operational semantics of event structure construction $S \xrightarrow{e} S'$ the proof then proceeds using the standard *simulation* argument. Moiseenko et al. [2020, §4.2] define a *simulation relation* $\mathcal{I}(P, T, G, TC, S, X)$ between the program P, a set of thread identifiers $T \subseteq \mathsf{Tid}$, 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 $\mathcal{I}(P, \mathtt{tid}(P), G, TC_{\mathrm{init}}(G), S_{\mathrm{init}}(P), \{e_0\})$ holds where*

- *$\mathtt{tid}(P)$ the set of threads of program P;*
- *$TC_{\mathrm{init}}(G) \triangleq \langle e_0, e_0\rangle$ is initial traversal configuration containing only the initialization event;*
- *$S_{\mathrm{init}}(P)$ is initial event structure also containing only the initialization event.*

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

Lemma F.4 (Simulation End). *If $\mathcal{I}(P, \mathtt{tid}(P), G, TC_{\mathrm{final}}(G), S, X)$ holds, where $TC_{\mathrm{final}}(G) \triangleq \langle G.\mathsf{E}, G.\mathsf{E}\rangle$ 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.

$$X' \triangleq X \setminus Br_{\text{prev}} \cup Br_{\text{cert}}$$
$$Br_{\text{prev}} \triangleq X|_t \triangleq \{e \in X \mid S.\mathtt{tid}(e) = t\}$$

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 $\mathcal{I}$. We present here only some of invariants derived from $\mathcal{I}$ 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 paperhttps://github.com/weakmemory/weakestmoToImm/.

The simulation relation $\mathcal{I}(P, T, G, TC, S, X)$ establishes a connection between the event structure S and execution graph G with the help of a function $\mathtt{s2g} : S.\mathtt{E} \to G.\mathtt{E}$ which maps an event of S to an event of G. This function can be lifted to sets of events in the following way[7]:

$$\text{for } A_S \subseteq S.\mathtt{E} : \lceil\!\lceil A_S \rceil\!\rceil \triangleq \{\mathtt{s2g}(e) \in G.\mathtt{E} \mid e \in A_S\}$$
$$\text{for } A_G \subseteq G.\mathtt{E} : \lfloor\!\lfloor A_G \rfloor\!\rfloor \triangleq \{e \in S.\mathtt{E} \mid \mathtt{s2g}(e) \in A_G\}$$

The simulation relation $\mathcal{I}$ 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:
   - $\lceil\!\lceil S.\mathtt{E}|_T \rceil\!\rceil = \lceil\!\lceil X \rceil\!\rceil = C \cup dom(G.\mathtt{po}^? ; [I])$

(2) Labels of events in S match labels of events in G modulo their values.
   (a) $\forall e \in S.\mathtt{E}.\ S.\{\mathtt{tid}, \mathtt{typ}, \mathtt{loc}, \mathtt{mod}\}(e) = G.\{\mathtt{tid}, \mathtt{typ}, \mathtt{loc}, \mathtt{mod}\}(\lceil\!\lceil e \rceil\!\rceil)$
   Labels of determined events match precisely.
   (b) $\forall e \in X \cap \lfloor\!\lfloor G.D_{\langle C,I \rangle} \rfloor\!\rfloor.\ S.\mathtt{val}(e) = G.\mathtt{val}(\lceil\!\lceil e \rceil\!\rceil)$

(3) Program order in S corresponds to program order in G:
   - $\lceil\!\lceil S.\mathtt{po} \rceil\!\rceil \subseteq G.\mathtt{po}$

(4) Identity relation in G corresponds to identity or conflict relation in S:
   - $\lfloor\!\lfloor \mathtt{id} \rfloor\!\rfloor \subseteq S.\mathtt{cf}^?$

(5) Reads in S are justified by writes that have already been observed by the corresponding events in G.
   (a) $\lceil\!\lceil S.\mathtt{jf} \rceil\!\rceil \subseteq G.\mathtt{rf}^? ; G.\mathtt{hb}^?$
   Moreover, restriction of justified-from relation on X corresponds to *stable justification relation* (see F.3) of the current traversal configuration in graph.

[7] In a similar manner it is possible to lift s2g to binary relations on events.

(b) $\lceil\!\lceil S.\mathtt{jf} \,;\, [X] \rceil\!\rceil \subseteq G.\mathtt{sjf}_{TC}$
As a consequence it is possible to derive that justification for covered events in X corresponds to their justification in G: $\lceil\!\lceil S.\mathtt{jf} \,;\, [X \cap \lfloor\!\lfloor C \rfloor\!\rfloor] \rceil\!\rceil \subseteq G.\mathtt{rf}$.
Finally, only issued events can be used for external justification.
(c) $dom(S.\mathtt{jfe}) \subseteq dom(S.\mathtt{ew}^* \,;\, [X \cap \lfloor\!\lfloor I \rfloor\!\rfloor])$

(6) Equivalent writes in S are mapped to the same write in G.
(a) $\lceil\!\lceil S.\mathtt{ew} \rceil\!\rceil \subseteq \mathtt{id}$
Also, every $S.\mathtt{ew}^*$ equivalence class has a representative issued event in X.
(b) $S.\mathtt{ew}^* \subseteq (S.\mathtt{ew}^* \,;\, [X \cap \lfloor\!\lfloor I \rfloor\!\rfloor] \,;\, S.\mathtt{ew}^*)^?$

(7) Coherence edges in S correspond to coherence or identity edges in G.
(a) $\lceil\!\lceil S.\mathtt{co} \rceil\!\rceil \subseteq G.\mathtt{co}^?$
Coherence edges in S ending in X restricted to threads T correspond precisely to coherence edges in G.
(b) $\lceil\!\lceil S.\mathtt{co} \,;\, [X|_T] \rceil\!\rceil \subseteq G.\mathtt{co}$

(8) Synchronize-with and happens-before relations in S conform to the corresponding relations in G.
(a) $\lceil\!\lceil S.\mathtt{sw} \rceil\!\rceil \subseteq G.\mathtt{sw}$
(b) $\lceil\!\lceil S.\mathtt{hb} \rceil\!\rceil \subseteq G.\mathtt{hb}$

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 1 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 $S.\mathtt{co}$, $S.\mathtt{sw}$, and $S.\mathtt{hb}$ 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 $G.D_{\langle C,I\rangle}$ (to be defined later in F.3), while Moiseenko et al. [2020] uses covered and issued events. As will become evident later $C \cup I \subseteq G.D_{\langle C,I\rangle}$ 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 $G.\mathtt{tid}(e)$ 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, $G.D_{\langle C,I\rangle}$, are events that must have the same label in G and in configuration X of S.

$$\begin{aligned} G.D_{\langle C,I\rangle} \triangleq\ & C \cup I \cup \\ & \cup\, G.\mathtt{W} \setminus \mathit{rng}(G.\mathtt{ppo}) \cup \\ & \cup\, \mathit{dom}(G.\mathtt{rfi}^? ; G.\mathtt{ppo} ; [I]) \cup \\ & \cup\, \mathit{rng}([I] ; G.\mathtt{rfi}) \cup \\ & \cup\, \mathit{rng}(G.\mathtt{rfe} ; [G.\mathtt{E}^{\sqsupseteq \mathsf{acq}}]) \end{aligned}$$

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

$$G.\mathtt{vf}_{\langle C,I\rangle} \triangleq [G.\mathtt{W}] ; (G.\mathtt{rf} ; [C])^? ; G.\mathtt{hb}^? \cup G.\mathtt{rf} ; [G.D_{\langle C,I\rangle}] ; G.\mathtt{po}^?$$

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

$$G.\mathtt{sjf}_{TC} \triangleq ([G.\mathtt{W}] ; (G.\mathtt{vf}_{TC} \cap =_{G.\mathtt{loc}}) ; [G.\mathtt{R}]) \setminus (G.\mathtt{co} ; G.\mathtt{vf}_{TC})$$

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.

$$G.D_{\langle C,I\rangle} \triangleq C \cup I \cup \mathit{dom}(G.\mathtt{rfi}^? ; G.\mathtt{ppo} ; [I]) \cup \mathit{rng}([I] ; G.\mathtt{rfi}) \cup \mathit{rng}(G.\mathtt{rfe} ; [G.\mathtt{E}^{\sqsupseteq \mathsf{acq}}])$$

The only difference of the modified version is that it also includes $G.\mathtt{W} \setminus \mathit{rng}(G.\mathtt{ppo})$, 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. *$G.\mathtt{vf}_{TC}; G.\mathtt{eco}$ is irreflexive.*

Proposition F.6. *$G.\mathtt{sjfi}_{TC} = G.\mathtt{rfi}$*

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

$$G.\mathtt{sjf}_{TC'}; [G.D_{TC}] = G.\mathtt{rf}; [G.D_{TC}] = G.\mathtt{sjf}_{TC}; [G.D_{TC}]$$

Proposition F.8. *Given a traversal step $G \vdash \langle C, I\rangle \longrightarrow \langle C', I'\rangle$, we have that*

$$G.\mathtt{sjf}_{\langle C',I'\rangle} = [I] ; G.\mathtt{sjfe}_{\langle C',I'\rangle} \cup G.\mathtt{sjfi}_{\langle C',I'\rangle}$$

Put simply, each stable external justification write is issued.

Proposition F.9. *Given a traversal step $G \vdash \langle C, I\rangle \longrightarrow_t \langle C', I'\rangle$, we have that*

$$G.\mathtt{sjf}_{\langle C',I'\rangle} ; [G.E \setminus G.E|_t] = G.\mathtt{sjf}_{\langle C,I\rangle} ; [G.E \setminus G.E|_t]$$

F.4 Simulation Relation for Certification Branch

Along with the simulation relation $\mathcal{I}(P, T, G, TC, S, X)$, the authors also define an auxiliary relation $\mathcal{I}^{\text{cert}}(P, T, G, TC, TC', S, X, Br_{\text{cert}})$, 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 $\mathcal{I}(P, T, G, TC', S', X')$ holds. Below we list core invariants of $\mathcal{I}^{\text{cert}}(P, T, G, \langle C, I\rangle, \langle C', I'\rangle, S, X, Br_{\text{cert}})$. 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) $\mathcal{I}(P, T \setminus t, G, \langle C, I\rangle, S, X)$ holds.
(2) $G \vdash \langle C, I\rangle \rightarrow_t \langle C', I'\rangle$, 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) $Br_{\text{cert}} \cap \lfloor\lfloor C \rfloor\rfloor = X|_t \cap \lfloor\lfloor C \rfloor\rfloor$.
  In general, set of events of the certification branch is a subset of covered/issued events and their po-predecessors from the new traversal configuration.
  (b) $\lceil Br_{\text{cert}} \rceil \subseteq C' \cup dom(G.\text{po}^? ; [I'])$.
(4) Labels of determined events in Br_{cert} match labels of events in G.
  (a) $\forall e \in Br_{\text{cert}} \cap \lfloor\lfloor G.D_{\langle C', I'\rangle} \rfloor\rfloor.\ S.\texttt{val}(e) = G.\texttt{val}(\lceil e \rceil)$
(5) Reads in Br_{cert} are justified according to the stable justification relation.
  (a) $\lceil S.\texttt{jf} ; [Br_{\text{cert}}] \rceil \subseteq G.\texttt{sjf}_{TC'}$
(6) For every issued event in Br_{cert} there exists $S.\text{ew}$ equivalent event in X.
  (a) $Br \cap \lfloor\lfloor I \rfloor\rfloor \subseteq dom(S.\text{ew}^? ; [X])$
  Symmetrically, every issued event in X within the processed part of the certification branch has an $S.\text{ew}$ equivalent event in Br_{cert}.
  (b) $X \cap \lfloor\lfloor I \cap \lceil Br_{\text{cert}} \rceil \rfloor\rfloor \subseteq dom(S.\text{ew}^? ; [Br_{\text{cert}}])$
(7) The $S.\text{co}$ edges ending in Br_{cert} corresponds to $G.\text{co}$ edges.
  (a) $\lceil S.\text{co} ; [Br_{\text{cert}}] \rceil \subseteq G.\text{co}$
  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) $\lceil S.\text{co} ; [X|_t \setminus \lfloor\lfloor \lceil Br_{cert} \rceil \rfloor\rfloor] \rceil \subseteq G.\text{co}$

F.5 Adapting the Proof

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

$$(\texttt{jf} \setminus (\texttt{jf}^? ; \texttt{hb})) ; \text{po} ; \text{ew} \subseteq \texttt{jf} ; (\text{po} \cup \texttt{lbpat})$$

Essentially, we need to show that if the simulation relation $\mathcal{I}(P, T, G, TC, S, X)$ 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_{cert}, such that $\mathcal{I}(P, T, G, TC', S', X')$ holds and S' also satisfies NO-BAIT-AND-SWITCH.

Moiseenko et al. [2020, §4.3.3] describes how to construct a single branch Br_{cert} and S' such that $\mathcal{I}(P, T, G, TC', S', X')$ holds, where S' is equal to original event structure S extended with the new events from Br_{cert}. The branch Br_{cert} is built with the help of the receptiveness property using $G.\texttt{sjf}_{TC'}$ to chose justification writes for read events. Unfortunately, if we take just this branch Br_{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 $\{Br_1, \dots Br_{n+1} = Br_{\text{cert}}\}$, where each branch is guided by a variant of the stable justification relation $G.\texttt{sjf}^i_{TC}$ (to be defined later). We can then show that each corresponding event structure S'_i satisfies NO-BAIT-AND-SWITCH and a weaker version of the simulation relation $\mathcal{I}^i_{\textbf{weak}}(P, G, TC, TC', S'_i, X'_i)$. We then demonstrate that the final event structure S'_{n+1} in this series satisfies the main simulation relation $\mathcal{I}(P, T, G, TC', S'_{n+1}, X'_{n+1})$.

The weak version of the simulation relation $\mathcal{I}^i_{\textbf{weak}}(P, T, G, TC, TC', S, X)$ is parameterized by the natural number i and two consecutive traversal configurations $G \vdash TC \longrightarrow TC'$. It has the following differences comparing to the regular simulation relation $\mathcal{I}(P, T, G, TC, S, X)$.

- In 2b instead of $G.D_{\langle C,I\rangle}$ we take $G.D^i_{\langle C',I'\rangle}$.
  - $\forall e \in X \cap \lfloor G.D^i_{\langle C',I'\rangle} \rfloor.\ S.\texttt{val}(e) = G.\texttt{val}(\lceil e \rceil)$
- In 5b instead of $G.\texttt{sjf}_{TC}$ we take $G.\texttt{sjf}^i_{TC'}$.
  - $\lceil S.\texttt{jf} ; [X] \rceil \subseteq G.\texttt{sjf}^i_{TC'}$

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 $G.\texttt{ppo}$ order. It means that the presence of w in certification branch and its label depend on the labels of these reads. The definition of $G.\texttt{sjf}_{TC'}$ 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 $G.\texttt{ppo}$. Changing justification for multiple reads at once violates NO-BAIT-AND-SWITCH, because the previous branch Br_{prev} of S and the certification branch Br_{cert} would have different set of incoming justification writes. It is possible, however, to replace justification writes for $G.\texttt{ppo}$-preceding reads *incrementally* one-by-one, constructing a series of certification branches $\{Br_1, \dots, Br_{n+1}\}$. 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 $\texttt{lbpat}$. Note that the newly constructed branches do not alter justification writes for reads $G.\texttt{ppo}$-preceding previously issued writes I, because $dom(G.\texttt{ppo} ; [I]) \subseteq G.D_{\langle C,I\rangle}$ and $G.\texttt{sjf}_{TC'}; [G.D_{TC}] = G.\texttt{sjf}_{TC}; [G.D_{TC}]$. In other words, all branches $\{Br_1, \dots, Br_{n+1}\}$ 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.*

$$G.\mathtt{R} \cap G.D_{\langle C,I\rangle} = G.\mathtt{R} \cap C \cup dom(G.\texttt{ppo} ; [I]) \cup rng([I] ; G.\texttt{rfi}) \cup rng(G.\texttt{rfe} ; [G.\mathtt{E}^{\sqsupseteq \mathsf{acq}}])$$

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

- $G.\mathtt{R} \cap G.\mathtt{W} = \emptyset$;
- $I \subseteq G.\mathtt{W}$;
- $dom(G.\texttt{rfi}) \subseteq G.\mathtt{W}$.

□

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

$$G.\texttt{vf}_{\langle C,I\rangle} \triangleq [G.\mathtt{W}] ; (G.\texttt{rf} ; [C])^? ; G.\texttt{hb}^? \cup G.\texttt{rf} ; [dom(G.\texttt{ppo} ; [I])] ; G.\texttt{po}^? \cup G.\texttt{rfe} ; [G.\mathtt{E}^{\sqsupseteq \mathsf{acq}}] ; G.\texttt{po}^?$$

PROOF. Note that $rng(G.\texttt{rf}) \subseteq \mathtt{R}$ and thus $G.\texttt{rf} ; [G.D_{\langle C,I\rangle}] = G.\texttt{rf} ; [\mathtt{R} \cap G.D_{\langle C,I\rangle}]$. Using Lemma F.10 we can rewrite $G.\texttt{vf}_{\langle C,I\rangle}$ as follows.

$$G.\texttt{vf}_{\langle C,I\rangle} = [G.\mathtt{W}] ; (G.\texttt{rf} ; [C])^? ; G.\texttt{hb}^? \cup$$

$$\cup\, G.\mathtt{rf} \,;\, [C] \,;\, G.\mathsf{po}^{?} \cup$$
$$\cup\, G.\mathtt{rf} \,;\, [dom(G.\mathsf{ppo} \,;\, [I])] \,;\, G.\mathsf{po}^{?} \cup$$
$$\cup\, G.\mathtt{rf} \,;\, [rng([I] \,;\, G.\mathtt{rfi})] \,;\, G.\mathsf{po}^{?} \cup$$
$$\cup\, G.\mathtt{rf} \,;\, [rng(G.\mathtt{rfe} \,;\, [G.\mathtt{E}^{\sqsupseteq \mathsf{acq}}])] \,;\, G.\mathsf{po}^{?}$$

Applying the following (in)equalities we arrive to the conclusion.

$$G.\mathtt{rf} \,;\, [C] \,;\, G.\mathsf{po}^{?} \subseteq [G.\mathtt{W}] \,;\, (G.\mathtt{rf} \,;\, [C])^{?} \,;\, G.\mathsf{hb}^{?}$$

$$G.\mathtt{rf} \,;\, [rng([I] \,;\, G.\mathtt{rfi})] \,;\, G.\mathsf{po}^{?} \subseteq [I] \,;\, G.\mathtt{rfi} \,;\, G.\mathsf{po}^{?} \subseteq [G.\mathtt{W}] \,;\, G.\mathsf{po} \subseteq [G.\mathtt{W}] \,;\, (G.\mathtt{rf} \,;\, [C])^{?} \,;\, G.\mathsf{hb}^{?}$$

$$G.\mathtt{rf} \,;\, [rng(G.\mathtt{rfe} \,;\, [G.\mathtt{E}^{\sqsupseteq \mathsf{acq}}])] \,;\, G.\mathsf{po}^{?} = G.\mathtt{rfe} \,;\, [G.\mathtt{E}^{\sqsupseteq \mathsf{acq}}] \,;\, G.\mathsf{po}^{?}$$

□

Lemma F.12. *Let $G \vdash \langle C, I\rangle \to \langle C', I'\rangle$. Then we have:*

$$\begin{aligned} G.D_{\langle C', I'\rangle} = G.D_{\langle C, I\rangle} &\cup \\ &\cup\, C' \setminus C \cup I' \setminus I \cup \\ &\cup\, dom(G.\mathtt{rfi}^{?} \,;\, G.\mathsf{ppo} \,;\, [I' \setminus I]) \cup \\ &\cup\, rng([I' \setminus I] \,;\, G.\mathtt{rfi}) \end{aligned}$$

Proof. Follows immediately from the definition of determined events and the monotonicity of the traversal step. □

Lemma F.13. *Let $G \vdash \langle C, I\rangle \to_t \langle C', I'\rangle$. Then we have:*

$$\begin{aligned} G.\mathtt{vf}_{\langle C', I'\rangle} = G.\mathtt{vf}_{\langle C, I\rangle} &\cup \\ &\cup\, [G.\mathtt{W}] \,;\, (G.\mathtt{rf} \,;\, [C' \setminus C])^{?} \,;\, G.\mathsf{hb}^{?} \cup \\ &\cup\, G.\mathtt{rf} \,;\, [dom(G.\mathsf{ppo} \,;\, [I' \setminus I])] \,;\, G.\mathsf{po}^{?} \cup \end{aligned}$$

Proof. Follows from the Lemma F.11 and the monotonicity of traversal step. □

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

$$dom(G.\mathtt{rfi} \,;\, G.\mathsf{ppo} \,;\, [I]) \subseteq G.\mathtt{W} \setminus rng(G.\mathsf{ppo}) \cup rng([dom(G.\mathsf{ppo} \,;\, [I])] \,;\, G.\mathsf{ppo})$$

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

$$\langle r_w, w_{\mathsf{iss}}\rangle \in G.\mathsf{ppo} \,;\, G.\mathtt{rfi} \,;\, G.\mathsf{ppo} \,;\, [I] \subseteq G.\mathsf{ppo} \,;\, [I]$$

(the last inequality follows from the definition of $G.\mathsf{ppo}$).
Therefore we have that $w \in rng([dom(G.\mathsf{ppo} \,;\, [I])] \,;\, G.\mathsf{ppo})$. □

Definition F.15. Let $G \vdash \langle C, I\rangle \to \langle C', I'\rangle$ and $\mathcal{E} \triangleq G.\mathtt{R} \cap (C' \setminus C) \cup dom(G.\mathsf{ppo} \,;\, [I' \setminus I])$.
Since ppo $\subseteq$ po we have that all events from $\mathcal{E}$ belong to the same thread, i.e., there exists $t \in \mathsf{Tid}$ such that for all $e \in \mathcal{E}$ we have $G.\mathtt{tid}(e) = t$. Therefore all the events in $\mathcal{E}$ can be ordered according to their po order. That is $\mathcal{E} = \{e_0, \ldots, e_n\}$, such that $\langle e_i, e_{i+1}\rangle \in \mathsf{po}$ for all $0 \le i \le n$.
Then we define $G.D^{i}_{\langle C', I'\rangle}$, $G.\mathtt{vf}^{i}_{\langle C', I'\rangle}$ and $G.\mathtt{sjf}^{i}_{\langle C', I'\rangle}$ as follows.

- $G.D^0_{\langle C',I'\rangle} = G.D_{\langle C,I\rangle}$
- $G.D^{i+1}_{\langle C',I'\rangle} = G.D^i_{\langle C',I'\rangle} \cup [e_i] \cup dom(G.\mathtt{rfi} ; G.\mathtt{ppo} ; [I' \setminus I]) \cap rng([e_i] ; G.\mathtt{ppo})$ for $0 < i < n$
- $G.D^{n+1}_{\langle C',I'\rangle} = G.D^n_{\langle C',I'\rangle} \cup [e_n] \cup rng([I' \setminus I] ; G.\mathtt{rfi})$
- $G.\mathtt{vf}^0_{\langle C',I'\rangle} \triangleq G.\mathtt{vf}_{\langle C,I\rangle}$
- $G.\mathtt{vf}^{i+1}_{\langle C',I'\rangle} \triangleq G.\mathtt{vf}^i_{\langle C',I'\rangle} \cup G.\mathtt{rf} ; [e_i] ; G.\mathtt{po}^?$ for $0 < i \leq n+1$
- $G.\mathtt{sjf}^i_{TC'} \triangleq ([G.\mathtt{W}] ; (G.\mathtt{vf}^i_{TC'} \cap =_{G.\mathtt{loc}}) ; [G.\mathtt{R}]) \setminus (G.\mathtt{co} ; G.\mathtt{vf}^i_{TC'})$ for $0 \leq i \leq n+1$

Lemma F.16. *We have that:*

- $G.D^{n+1}_{\langle C',I'\rangle} = G.D_{\langle C',I'\rangle}$
- $G.\mathtt{vf}^{n+1}_{\langle C',I'\rangle} = G.\mathtt{vf}_{\langle C',I'\rangle}$
- $G.\mathtt{sjf}^{n+1}_{\langle C',I'\rangle} = G.\mathtt{sjf}_{\langle C',I'\rangle}$

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

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

Proof. By Prop. F.5 we know that $G.\mathtt{vf}_{\langle C',I'\rangle} ; G.\mathtt{eco}$ is irreflexive. But since $G.\mathtt{vf}^i_{\langle C',I'\rangle} \subseteq G.\mathtt{vf}$ we arrive at the conclusion that $G.\mathtt{vf}^i_{\langle C',I'\rangle} ; G.\mathtt{eco}$ should also be irreflexive. □

Lemma F.18. *For* $0 \leq i \leq n$ *let* $\mathcal{R}_i \triangleq rng([e_i] ; (\mathtt{po} \cap\ =_{\mathtt{loc}})^? ; [\mathtt{R}])$.
Then the following equation holds.

$$G.\mathtt{sjf}^{i+1}_{TC'} ; [G.\mathtt{E} \setminus \mathcal{R}_i] = G.\mathtt{sjf}^i_{TC'} ; [G.\mathtt{E} \setminus \mathcal{R}_i]$$

Proof. By definition of $G.\mathtt{sjf}^{i+1}_{TC'}$ we have:

$$G.\mathtt{sjf}^{i+1}_{TC'} ; [G.\mathtt{E} \setminus \mathcal{R}_i] = ([G.\mathtt{W}] ; (G.\mathtt{vf}^{i+1}_{TC'} \cap =_{G.\mathtt{loc}}) ; [G.\mathtt{R}]) \setminus (G.\mathtt{co} ; G.\mathtt{vf}^{i+1}_{TC'}) ; [G.\mathtt{E} \setminus \mathcal{R}_i] =$$
$$= ([G.\mathtt{W}] ; ((G.\mathtt{vf}^{i+1}_{TC'} \cap =_{G.\mathtt{loc}}) ; [G.\mathtt{R} \setminus \mathcal{R}_i])) \setminus (G.\mathtt{co} ; (G.\mathtt{vf}^{i+1}_{TC'} \cap =_{G.\mathtt{loc}}) ; [G.\mathtt{R} \setminus \mathcal{R}_i])$$

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

$$G.\mathtt{vf}^{i+1}_{TC'} ; [G.\mathtt{E} \setminus \mathcal{R}_i] = (G.\mathtt{vf}^i_{\langle C',I'\rangle} \cap =_{G.\mathtt{loc}}) ; [G.\mathtt{E} \setminus \mathcal{R}_i] \cup G.\mathtt{rf} ; [e_i] ; (G.\mathtt{po}^? \cap =_{G.\mathtt{loc}}) ; [G.\mathtt{E} \setminus \mathcal{R}_i]$$

Then it is easy to see that $G.\mathtt{rf} ; [e_i] ; (G.\mathtt{po}^? \cap =_{G.\mathtt{loc}}) ; [G.\mathtt{E} \setminus \mathcal{R}_i] = \emptyset$. □

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

$$G.\mathtt{sjf}^{i+1}_{TC'} ; [G.D^i_{TC}] = G.\mathtt{sjf}^i_{TC'} ; [G.D^i_{TC'}]$$

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

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

$$G.\mathtt{sjf}^{i+1}_{\langle C',I'\rangle} ; [e_i] = G.\mathtt{rf}_{\langle C',I'\rangle} ; [e_i]$$

Or in other words, stable justification for event e_i *coincides with the reads-from relation.*

Proof. Since $G.\mathtt{sjf}$ is functional and complete for read events it is sufficient to prove

$$G.\mathtt{rf}_{\langle C',I'\rangle} ; [e_i] \subseteq G.\mathtt{sjf}^{i+1}_{\langle C',I'\rangle} ; [e_i]$$

It is easy to see that $G.\mathtt{rf}_{\langle C',I'\rangle} ; [e_i] \subseteq G.\mathtt{vf}^{i+1}_{\langle C',I'\rangle} ; [e_i]$. Let $\langle w, e_i\rangle \in G.\mathtt{rf}_{\langle C',I'\rangle} ; [e_i]$. Then it is left to show that $\langle w, e_i\rangle \notin G.\mathtt{co} ; G.\mathtt{vf}^{i+1}_{\langle C',I'\rangle}$. Suppose otherwise. Then there exists w' s.t. $\langle w, w'\rangle \in G.\mathtt{co}$ and $\langle w', e_i\rangle \in G.\mathtt{vf}^{i+1}_{\langle C',I'\rangle}$. It cannot be the case that $\langle w', e_i\rangle \in G.\mathtt{rf} ; [e_i] ; G.\mathtt{po}^?$ so therefore $\langle w', e_i\rangle \in G.\mathtt{vf}^{i}_{\langle C',I'\rangle}$. But we also have $\langle e_i, w'\rangle \in G.\mathtt{fr} \subseteq G.\mathtt{eco}$ which contradicts Lemma F.17. □

Lemma F.21. *For any $0 \le i \le n+1$ we have the following.*

$$G.\mathtt{sjf}^{i}_{TC'} ; [G.D^i_{TC}] = G.\mathtt{rf} ; [G.D^i_{TC'}]$$

Proof. Proof goes straightforwardly by induction on i using Prop. F.7 and Lemmas F.19 and F.20. □

Lemma F.22. *Let $0 \le i \le n$. Then we have that:*

$$G.\mathtt{sjf}^{i+1}_{\langle C',I'\rangle} ; [rng([e_i] ; G.\mathtt{po} \cap =_{G.\mathtt{loc}})] \subseteq G.\mathtt{sjf}^{i}_{\langle C',I'\rangle} \cup G.\mathtt{sjf}^{i+1}_{\langle C',I'\rangle} ; [e_i] ; G.\mathtt{po}$$

Proof. Consider $\langle w, r\rangle \in G.\mathtt{sjf}^{i+1}_{\langle C',I'\rangle} \subseteq G.\mathtt{vf}^{i+1}_{\langle C',I'\rangle} = G.\mathtt{vf}^{i}_{\langle C',I'\rangle} \cup G.\mathtt{rf} ; [e_i] ; G.\mathtt{po}^?$. If $\langle w, r\rangle \in G.\mathtt{rf} ; [e_i] ; G.\mathtt{po}^?$ then by Lemma F.20 $\langle w, r\rangle \in G.\mathtt{sjf}^{i+1}_{\langle C',I'\rangle} ; [e_i] ; G.\mathtt{po}^?$ and we are done. Otherwise $\langle w, r\rangle \in G.\mathtt{vf}^{i}_{\langle C',I'\rangle}$. But then

$$\langle w, r\rangle \in G.\mathtt{vf}^{i}_{\langle C',I'\rangle} \setminus G.\mathtt{co} ; \mathtt{vf}^{i+1}_{\langle C',I'\rangle} \subseteq G.\mathtt{vf}^{i}_{\langle C',I'\rangle} \setminus G.\mathtt{co} ; \mathtt{vf}^{i}_{\langle C',I'\rangle} = G.\mathtt{sjf}^{i}_{\langle C',I'\rangle}$$

□

Lemma F.23. *For any $0 \le i \le n$ the following is true.*

$$G.\mathtt{sjf}^{i}_{\langle C',I'\rangle} ; [e_i] \subseteq G.\mathtt{sjf}^{i}_{\langle C',I'\rangle} ; G.\mathtt{hb}$$

Proof. Let $\langle w, e_i\rangle \in G.\mathtt{sjf}^{i}_{\langle C',I'\rangle} \subseteq G.\mathtt{vf}^{i}_{\langle C',I'\rangle}$. By Lemma F.11:

$$G.\mathtt{vf}^{i}_{\langle C',I'\rangle} = [G.\mathtt{W}];(G.\mathtt{rf};[C])^?;G.\mathtt{hb}^?\cup G.\mathtt{rf};[dom(G.\mathtt{ppo} ; [e_0, \ldots, e_{i-1}])];G.\mathtt{po}^?\cup G.\mathtt{rfe};[G.\mathtt{E}^{\sqsupseteq \mathtt{acq}}];G.\mathtt{po}^?$$

It is left to notice the following facts

- $G.\mathtt{rf} ; [C] = G.\mathtt{sjf}^{i}_{\langle C',I'\rangle} ; [C]$;
- $G.\mathtt{rf} ; [e_j] = G.\mathtt{sjf}^{i}_{\langle C',I'\rangle} ; [e_j]$ for all $j < i$ (by Lemma F.21);
- $G.\mathtt{rfe} ; [G.\mathtt{E}^{\sqsupseteq \mathtt{acq}}] = G.\mathtt{sjfe}^{i}_{\langle C',I'\rangle} ; [G.\mathtt{E}^{\sqsupseteq \mathtt{acq}}]$ (also by Lemma F.21).

□

Lemma F.24. *Let $0 \le i \le n$. Then we have that:*

$$G.\mathtt{sjf}^{i+1}_{TC'} \setminus ((G.\mathtt{sjf}^{i+1}_{TC'})^? ; G.\mathtt{hb}) ; [rng([e_i] ; G.\mathtt{po} \cap =_{G.\mathtt{loc}})] =$$
$$= G.\mathtt{sjf}^{i}_{TC'} \setminus ((G.\mathtt{sjf}^{i}_{TC'})^? ; G.\mathtt{hb}) ; [rng([e_i] ; G.\mathtt{po} \cap =_{G.\mathtt{loc}})]$$

Proof. From left to right $\subseteq$ it follows from the Lemma F.22. In the opposite direction, consider w, r s.t.

- $\langle w, r\rangle \in G.\mathtt{sjf}^{i}_{TC'}$;
- $\langle w, r\rangle \notin (G.\mathtt{sjf}^{i}_{TC'})^? ; G.\mathtt{hb}$;
- $r \in rng([e_i] ; G.\mathtt{po} \cap =_{G.\mathtt{loc}})$.

Consider the write w' s.t. $\langle w', r\rangle \in G.\mathtt{sjf}^{i+1}_{TC'}$. By Lemma F.22 either

- $\langle w', r\rangle \in G.\mathtt{sjf}^{i}_{TC'}$ or

- $\langle w', r\rangle \in G.\mathtt{sjf}^{i+1}_{TC'}\,;[e_i]\,;G.\mathsf{po}$.

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

Lemma F.25. *For any $0 \le i \le n+1$ we have the following.*

$$G.\mathtt{sjfi}^{i}_{TC'} = G.\mathtt{rfi}$$

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

Otherwise consider the cases. If $r = e_i$ then by Lemma F.20 $G.\mathtt{sjfi}^{i+1}_{TC'}\,;[e_i] = G.\mathtt{rfi}\,;[e_i]$. If $r \in rng([e_i]\,;G.\mathsf{po}\cap =_{G.\mathtt{loc}})$ then by Lemma F.22 either

- $G.\mathtt{sjfi}^{i+1}_{TC'}\,;[r] = G.\mathtt{sjfi}^{i}_{TC'}\,;[r] = G.\mathtt{rfi}\,;[r]$ (the last equality due to inductive assumption), or
- $G.\mathtt{sjfi}^{i+1}_{TC'}\,;[r] \subseteq G.\mathtt{sjfi}^{i+1}_{TC'}\,;[e_i]\,;G.\mathsf{po}$, and given that $G.\mathtt{sjfi}^{i+1}_{TC'}\,;[e_i] = G.\mathtt{rfi}\,;[e_i]$ we conclude that $G.\mathtt{sjfi}^{i+1}_{TC'}\,;[r] = G.\mathtt{rfi}\,;[r]$

□

Lemma F.26. *For any $0 \le i \le n+1$ we have the following.*

$$G.\mathtt{sjf}^{i}_{TC'} = [I]\,;G.\mathtt{sjfe}^{i}_{TC'} \cup G.\mathtt{sjfi}^{i}_{TC'}$$

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

Lemma F.27. *For any $0 \le i \le n$ we have the following.*

$$G.\mathtt{sjf}^{i+1}_{TC'}\,;[G.E \setminus G.E|_t] = G.\mathtt{sjf}^{i}_{TC'}\,;[G.E \setminus G.E|_t]$$

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 $\mathcal{I}^{i}_{\mathbf{weak}}(P, T, G, TC, TC', S, X)$, we need to make some changes to the simulation relation for the certification branch and introduce $\mathcal{I}^{i+1\ \mathbf{cert}}_{\mathbf{weak}}(P, T, G, \langle C, I\rangle, \langle C', I'\rangle, S, X, Br_{i+1})$.

- In 1 instead of $\mathcal{I}(P, T \setminus t, G, \langle C, I\rangle, S, X)$ we take $\mathcal{I}^{i}_{\mathbf{weak}}(P, T \setminus t, G, \langle C, I\rangle, \langle C', I'\rangle, S, X)$
- 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.
  - $Br_{i+1} \cap \lfloor\!\lfloor G.\mathsf{po}\,;[e_i]\rfloor\!\rfloor = X|_t \cap \lfloor\!\lfloor G.\mathsf{po}\,;[e_i]\rfloor\!\rfloor$
- In 4a instead of $G.D_{\langle C', I'\rangle}$ we take $G.D^{i+1}_{\langle C', I'\rangle}$.
  - $\forall e \in Br_{i+1} \cap \lfloor\!\lfloor G.D^{i+1}_{\langle C', I'\rangle}\rfloor\!\rfloor.\ S.\mathtt{val}(e) = G.\mathtt{val}(\lceil\!\lceil e\rceil\!\rceil)$
- In 5a instead of $G.\mathtt{sjf}_{\langle C', I'\rangle}$ we take $G.\mathtt{sjf}^{i+1}_{\langle C', I'\rangle}$.
  - $\lceil\!\lceil S.\mathtt{jf}\,;[Br_{i+1}]\rceil\!\rceil \subseteq G.\mathtt{sjf}^{i+1}_{\langle C', I'\rangle}$

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

(1) Simulation relation $\mathcal{I}(P, T, G, TC, S, X)$ implies weak simulation relation $\mathcal{I}^{0}_{\mathbf{weak}}(P, T, G, TC, TC', S, X)$.

(2) Given that weak simulation relation holds, it is possible to start a construction of the next certification branch, i.e., from $\mathcal{I}^{i}_{\mathbf{weak}}(P,T,G,TC,TC',S,X)$ we can prove that $\mathcal{I}^{i+1\ \mathbf{cert}}_{\mathbf{weak}}(P,T,G,TC,TC',S,X,Br_{i+1})$ holds, where $Br_{i+1} = X|_t \cap \lceil\!\lceil G.\mathsf{po}\,;\,[e_i]\rceil\!\rceil$;
(3) It is possible to add an event to the certification branch preserving the simulation relation and NO-BAIT-AND-SWITCH, i.e., if $\mathcal{I}^{i\ \mathbf{cert}}_{\mathbf{weak}}(P,T,G,TC,TC',S,X,Br_i)$ holds and $S \xrightarrow{e} S'$, s.t. $dom(S'.\mathsf{po}\,;\,[e]) = Br_i$ then $\mathcal{I}^{i\ \mathbf{cert}}_{\mathbf{weak}}(P,T,G,TC,TC',S',X,Br_i \cup \{e\})$ and S' satisfies NO-BAIT-AND-SWITCH assuming that S also satisfies it.
(4) Given a complete certification branch it is possible to establish the weak simulation relation, i.e., from $\mathcal{I}^{i\ \mathbf{cert}}_{\mathbf{weak}}(P,T,G,TC,TC',S,X,Br_i)$ where $\lceil\!\lceil Br_i\rceil\!\rceil = C' \cup dom(G.\mathsf{po}^?\,;\,[I'])$ we can prove that $\mathcal{I}^{i}_{\mathbf{weak}}(P,T,G,TC,TC',S,X')$ holds, where $X' \triangleq X \setminus X|_t \cup Br_i$.
(5) Given the complete last certification branch it is possible to establish the regular simulation relation, i.e., from $\mathcal{I}^{n+1\ \mathbf{cert}}_{\mathbf{weak}}(P,T,G,TC,TC',S,X,Br_{n+1})$ where $\lceil\!\lceil Br_{n+1}\rceil\!\rceil = C' \cup dom(G.\mathsf{po}^?\,;\,[I'])$ we can prove that $\mathcal{I}(P,T,G,TC',S,X')$ holds, where $X' \triangleq X \setminus X|_t \cup Br_{n+1}$.

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. *$\mathcal{I}(P,G,TC,S,X)$ and $G \vdash TC \rightarrow_t TC'$ implies $\mathcal{I}^{0}_{\mathbf{weak}}(P,G,TC,TC',S,X)$.*

Proof. Holds trivially since by definition $G.D^{0}_{\langle C',I'\rangle} = G.D_{\langle C,I\rangle}$ and $G.\mathtt{sjf}^{0}_{\langle C',I'\rangle} = G.\mathtt{sjf}_{\langle C,I\rangle}$. □

Lemma F.29. *Suppose $\mathcal{I}^{i}_{\mathbf{weak}}(P,T,G,TC,TC',S,X)$ holds, and $G \vdash TC \rightarrow_t TC'$. Then $\mathcal{I}^{i+1\ \mathbf{cert}}_{\mathbf{weak}}(P,T,G,TC,TC',S,X,Br_{i+1})$ holds, where $Br_{i+1} = X|_t \cap \lfloor\!\lfloor dom(G.\mathsf{po}\,;\,[e_i])\rfloor\!\rfloor$.*

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

- 1 and 2 hold trivially by our assumptions.
- 3a holds trivially by our choice of Br_{i+1}:

$$Br_{i+1} \cap \lfloor\!\lfloor dom(G.\mathsf{po}\,;\,[e_i])\rfloor\!\rfloor = X|_t \cap \lfloor\!\lfloor dom(G.\mathsf{po}\,;\,[e_i])\rfloor\!\rfloor$$

- 3b holds since $dom(\mathsf{po}\,;\,[e_i]) \subseteq C' \cup dom(G.\mathsf{po}^?\,;\,[I'])$.
- To see why 4a holds first consider the following equation.

$$Br_{i+1} \cap \lfloor\!\lfloor G.D^{i+1}_{TC'}\rfloor\!\rfloor = X|_t \cap \lfloor\!\lfloor dom(G.\mathsf{po}\,;\,[e_i]) \cap G.D^{i+1}_{TC'}\rfloor\!\rfloor = X|_t \cap \lfloor\!\lfloor G.D^{i}_{TC}\rfloor\!\rfloor$$

  and for $e \in X \cap \lfloor\!\lfloor G.D^{i}_{TC}\rfloor\!\rfloor$ we know that $S.\mathtt{lab}(e) = G.\mathtt{lab}(\lceil\!\lceil e\rceil\!\rceil)$ by the property 4a of certification simulation relation.
- 5a holds because

$$\lceil\!\lceil S.\mathtt{jf}\,;\,[Br_{i+1}]\rceil\!\rceil = \lceil\!\lceil S.\mathtt{jf}\,;\,[X \cap \lfloor\!\lfloor dom(G.\mathsf{po}\,;\,[e_i])\rfloor\!\rfloor]\rceil\!\rceil \subseteq G.\mathtt{sjf}^{i}_{TC}\,;\,[dom(G.\mathsf{po}\,;\,[e_i])] \subseteq G.\mathtt{sjf}^{i+1}_{TC'}$$

  The last inequality follows from lemma F.18 and the fact that

$$dom(G.\mathsf{po}\,;\,[e_i]) \subseteq \mathcal{R}_i = G.E \setminus rng([e_i];(G.\mathsf{po} \cap =_{G.\mathtt{loc}})^?\,;\,[G.\mathsf{R}])$$

- 6a holds by our choice of Br_{i+1}:

$$Br_{i+1} \cap \lfloor\!\lfloor I\rfloor\!\rfloor = X|_t \cap \lfloor\!\lfloor dom(G.\mathsf{po}\,;\,[e_i])\rfloor\!\rfloor \cap \lfloor\!\lfloor I\rfloor\!\rfloor \subseteq dom(S.\mathtt{ew}^?\,;\,[X])$$

- 6b also holds by our choice of Br_{i+1}:

$$X \cap \lfloor I \cap \lceil Br_{i+1} \rceil \rfloor X \cap \lfloor I \cap \lceil X|_t \cap \lfloor dom(G.\mathsf{po} ; [e_i]) \rfloor \rceil \rfloor \subseteq$$
$$\subseteq X \cap \lfloor I \cap \lceil X|_t \rceil \cap dom(G.\mathsf{po} ; [e_i]) \rfloor \subseteq$$
$$\subseteq X|_t \cap \lfloor I \cap dom(G.\mathsf{po} ; [e_i]) \rfloor \subseteq Br_{i+1} \cap \lfloor I \rfloor \subseteq dom(S.\mathsf{ew}^? ; [Br_{i+1}])$$

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

□

Lemma F.30. *For $1 \le i \le n+1$ if $\mathcal{I}^{i\ \mathsf{cert}}_{\mathbf{weak}}(P, T, G, TC, TC', S, X, Br_i)$ holds and $S \xrightarrow{e} S'$, s.t.*

- $dom(S'.\mathsf{po} ; [e]) = Br_i$;
- $\lceil e \rceil \in C' \cup dom(G.\mathsf{po}^? ; [I'])$;
- $\lceil S'.\mathtt{jf} ; [e] \rceil \subseteq G.\mathtt{sjf}^i_{TC'}$

then $\mathcal{I}^{i\ \mathsf{cert}}_{\mathbf{weak}}(P, T, G, TC, TC', S', X, Br_i \cup \{e\})$ holds.

Proof. We need to show that each property of $\mathcal{I}^{i\ \mathsf{cert}}_{\mathbf{weak}}$ holds.

- To show 1 we refer to the proof of the similar statement in [Moiseenko et al. 2020]. Since the only difference between $\mathcal{I}(P, T \setminus t, G, \langle C, I\rangle, S, X)$ and $\mathcal{I}^i_{\mathbf{weak}}(P, T \setminus t, G, \langle C, I\rangle, \langle C', I'\rangle, S, X)$ is that the later uses $G.D^i_{\langle C', I'\rangle}$ and $G.\mathtt{sjf}^i_{\langle C', I'\rangle}$ instead of $G.D_{\langle C, I\rangle}$ and $G.\mathtt{sjf}_{\langle C, I\rangle}$ the proof proceeds in a similar vein. The only significant point we want to emphasize is the proof of 5c. To show that $dom(S'.\mathtt{jfe}) \subseteq dom(S.\mathsf{ew}^* ; [X \cap \lfloor I \rfloor])$ the original proof utilized the fact that $\lceil S'.\mathtt{jfe} ; [e] \rceil \subseteq G.\mathtt{sjf}_{\langle C', I'\rangle}$ and property F.8. We instead note that $\lceil S'.\mathtt{jfe} ; [e] \rceil \subseteq G.\mathtt{sjf}^i_{\langle C', I'\rangle}$ and use similar Lemma F.26.
- 2 follow immediately from $\mathcal{I}^{i\ \mathsf{cert}}_{\mathbf{weak}}(P, G, TC, TC', S, X, Br_i)$.
- 3a holds because

$$(Br_i \cup \{e\}) \cap \lfloor dom(G.\mathsf{po} ; [e_{i-1}]) \rfloor = Br_i \cap \lfloor dom(G.\mathsf{po} ; [e_{i-1}]) \rfloor = X|_t \cap \lfloor dom(G.\mathsf{po} ; [e_{i-1}]) \rfloor$$

  3b holds trivially by the precondition $\lceil e \rceil \in C' \cup dom(G.\mathsf{po}^? ; [I'])$.
- 4 and 5 hold by the choice of e, as we chose labels of events and their justification according to $G.\mathtt{sjf}^i_{TC'}$ and by lemma F.19 and proposition F.7 we have $G.\mathtt{sjf}^i_{TC'} ; [G.D^i_{TC'}] = G.\mathtt{rf} ; [G.D^i_{TC'}]$.
- To show 6 we note that $I \subseteq G.D_{\langle C, I\rangle} \subseteq G.D^i_{\langle C', I'\rangle}$ and thus if $\lceil e \rceil \in I$ then $S'.\mathtt{val}(e) = G.\mathtt{val}(\lceil e \rceil)$. Therefore, similarly as it is done in [Moiseenko et al. 2020], we can pick an equivalent write in $X|_t$, s.t. the labels of two writes would be equal. As we have shown above using the properties of $G.D^i_{\langle C', I'\rangle}$ both writes would have label equal to $G.\mathtt{val}(\lceil e \rceil)$.
- Since we update $S'.\mathtt{co}$ 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.

□

Lemma F.31. *For $1 \le i \le n$ if $\mathcal{I}^{i\ \mathsf{cert}}_{\mathbf{weak}}(P, T, G, TC, TC', S, X, Br_i)$ holds and $\lceil Br_i \rceil = C' \cup dom(G.\mathsf{po}^? ; [I'])$ then $\mathcal{I}^i_{\mathbf{weak}}(P, T, G, TC, TC', S, X')$ holds, where $X' \triangleq X \setminus X|_t \cup Br_i$.*

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

- First we note that 2a, 3, 4, 5a, 5c, 6, 7a, and 8 follow immediately from the same properties of $\mathcal{I}^{i-1}_{\mathbf{weak}}(P, T, G, TC, TC', S, X')$ which is implied from $\mathcal{I}^{i\ \mathsf{cert}}_{\mathbf{weak}}(P, T, G, TC, TC', S, X, Br_i)$ by 1.
- 1 holds trivially due to precondition $\lceil Br_i \rceil = C' \cup dom(G.\mathsf{po}^? ; [I'])$ and our choice of X'.

- 2b holds because we have

$$X' \cap \llbracket G.D^i_{\langle C', I'\rangle} \rrbracket = (X \setminus X|_t) \cap G.D^{i-1}_{\langle C', I'\rangle} \cup Br_i \cap \llbracket G.D^i_{\langle C', I'\rangle} \rrbracket$$

  For left subset we derive the required property from 2b of $\mathcal{I}^{i-1}_{\mathbf{weak}}(P, T, G, TC, TC', S, X')$, for the right subset we derive it from 4a of $\mathcal{I}^{i\ \mathbf{cert}}_{\mathbf{weak}}(P, T, G, TC, TC', S, X, Br_i)$.
- To see why 5b holds first note that.

$$\llbracket S.\mathtt{jf} ; [X'] \rrbracket = \llbracket S.\mathtt{jf} ; [X \setminus X|_t] \rrbracket \cup \llbracket S.\mathtt{jf} ; [Br_i] \rrbracket$$

  Then $\llbracket S.\mathtt{jf} ; [Br_i] \rrbracket \subseteq G.\mathtt{sjf}^i_{TC}$ is implied by 5a of $\mathcal{I}^{i\ \mathbf{cert}}_{\mathbf{weak}}(P, T, G, TC, TC', S, X, Br_i)$. Next using 5b of $\mathcal{I}^{i-1}_{\mathbf{weak}}(P, T, G, TC, TC', S, X')$ and Lemma F.27 we prove the following.

$$\llbracket S.\mathtt{jf} ; [X \setminus X|_t] \rrbracket \subseteq G.\mathtt{sjf}^{i-1} ; [G.E \setminus G.E|_t] \subseteq G.\mathtt{sjf}^i$$

- 7b can be proven by the similar reasoning using property 7a of $\mathcal{I}^{i\ \mathbf{cert}}_{\mathbf{weak}}(P, T, G, TC, TC', S, X, Br_i)$ and 7b of $\mathcal{I}^{i-1}_{\mathbf{weak}}(P, T, G, TC, TC', S, X')$.

□

Lemma F.32. *If $\mathcal{I}^{n+1\ \mathbf{cert}}_{\mathbf{weak}}(P, T, G, TC, TC', S, X, Br_{n+1})$ holds and $\llbracket Br_{n+1} \rrbracket = C' \cup dom(G.\mathtt{po}^? ; [I'])$ then $\mathcal{I}(P, T, G, TC', S, X')$ holds, where $X' \triangleq X \setminus X|_t \cup Br_{n+1}$.*

Proof. We note that due to lemma F.16 $\mathcal{I}^{n+1\ \mathbf{cert}}_{\mathbf{weak}}(P, T, G, TC, TC', S, X, Br_{n+1})$ is equivalent to $\mathcal{I}^{\mathbf{cert}}(P, T, G, TC, TC', S, X, Br_{n+1})$ 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 $S'.\mathtt{ew} = S.\mathtt{ew}$. Then S' also satisfies* NO-BAIT-AND-SWITCH.

Proof. Note that $S.\mathtt{E}$ is prefix-closed w.r.t. $\mathtt{po}$ and $\mathtt{jf}$, i.e., $S.\mathtt{po}' ; [S.\mathtt{E}] = S.\mathtt{po}$ and $S.\mathtt{jf}' ; [S.\mathtt{E}] = S.\mathtt{jf}$. Therefore $S'.\mathtt{jf} ; S'.\mathtt{po} ; S'.\mathtt{ew} = S.\mathtt{jf} ; S.\mathtt{po} ; S.\mathtt{ew}$. From that we can derive the following.

$$(S'.\mathtt{jf} \setminus (S'.\mathtt{jf}^? ; S'.\mathtt{hb})) ; S'.\mathtt{po} ; S'.\mathtt{ew} =$$
$$= (S.\mathtt{jf} \setminus (S.\mathtt{jf}^? ; S.\mathtt{hb})) ; S.\mathtt{po} ; S.\mathtt{ew} \subseteq S.\mathtt{jf} ; (S.\mathtt{po} \cup S.\mathtt{lbpat}) \subseteq S'.\mathtt{jf} ; (S'.\mathtt{po} \cup S'.\mathtt{lbpat})$$

The last inequality follows from the fact that $\mathtt{po}$, $\mathtt{jf}$ and $\mathtt{lbpat}$ are monotone w.r.t. addition of new events $S \xrightarrow{e} S'$. □

Lemma F.34. *For $0 \le i \le n$ if $\mathcal{I}^{i+1\ \mathbf{cert}}_{\mathbf{weak}}(P, T, G, TC, TC', S, X, Br_{i+1})$ holds, $S \xrightarrow{e} S'$, s.t.*

- *$dom(S'.\mathtt{po} ; [e]) = Br_{i+1}$;*
- *$\llbracket e \rrbracket \in C' \cup dom(G.\mathtt{po}^? ; [I'])$;*
- *$\llbracket S'.\mathtt{jf} ; [e] \rrbracket \subseteq G.\mathtt{sjf}^i_{TC'}$*

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 $e \in \mathtt{W}$. Otherwise by construction $S'.\mathtt{ew} = S.\mathtt{ew}$ and using Lemma F.33 we can immediately conclude that S' satisfies NO-BAIT-AND-SWITCH. So let $e \in \mathtt{W}$.

By the construction of [Moiseenko et al. 2020, §4.3.2] the $\mathtt{ew}$ 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., $w \in X \cap \llbracket I \rrbracket$ and $\llbracket e \rrbracket = \llbracket e \rrbracket$. In this case e is

attached to the $S.\mathsf{ew}^*$ equivalence class of w by adding a new 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 $Br_i = X|_t$ and Br_{i+1}. By Lemma F.30 we know that $\mathcal{I}^{i+1\ \mathbf{cert}}_{\mathbf{weak}}(P, T, G, TC, TC', S', X, Br_{i+1} \cup \{e\})$ holds and thus by property 5b of weak simulation relation and property 5a of simulation relation for certification branch we know that

- $[\![S.\mathsf{jf}\ ;\ [Br_i]]\!] \subseteq G.\mathsf{sjf}^i_{\langle C', I'\rangle}$, and
- $[\![S.\mathsf{jf}\ ;\ [Br_{i+1}]]\!] \subseteq G.\mathsf{sjf}^{i+1}_{\langle C', I'\rangle}$.

By Lemma F.18 we also know that:

$$G.\mathsf{sjf}^{i+1}_{TC'}\ ;\ [G.\mathsf{E} \setminus \mathcal{R}_i] = G.\mathsf{sjf}^{i}_{TC'}\ ;\ [G.\mathsf{E} \setminus \mathcal{R}_i]$$

where $\mathcal{R}_i \triangleq rng([e_i \cup rng([I' \setminus I]\ ;\ G.\mathsf{rfi})]\ ;\ (\mathsf{po} \cap =_{\mathtt{loc}})^?\ ;\ [\mathsf{R}])$. 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 for reads in $\mathcal{R}_i$.

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

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

But that means justification writes for reads in Br_{i+1} and Br_i corresponding to r should either be taken from $S'.\mathsf{jf}\ ;\ S'.\mathsf{hb}$ 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

$$[\![S'.\mathsf{jf}\ ;\ [r_i]]\!] = [\![S.\mathsf{jf}\ ;\ [r_i]]\!] \subseteq G.\mathsf{sjf}^{i}_{\langle C', I'\rangle}\ ;\ [e_i] \subseteq [G.\mathsf{W}]\ ;\ (G.\mathsf{rf}\ ;\ [C])^?\ ;\ G.\mathsf{hb}$$

But then it should also be the case that:

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

From that we can conclude that

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

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., $\langle r_{i+1}, w\rangle \in \mathtt{lbpat}$ (recall that w is an equivalent write we picked for e, i.e., $\langle w, e\rangle \in S'.\mathsf{ew}$). 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

$$\mathtt{lbpat} \triangleq \mathsf{cf}_{\mathtt{imm}}\ ;\ [rng(\mathsf{jf} \cap (\mathsf{jf}^?\ ;\ \mathsf{hb}))]\ ;\ \mathsf{po}$$

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: $\langle r_{i+1}, r_i\rangle \in S'.\mathsf{cf}_{\mathtt{imm}}$. Moreover, as we have shown r_i is locally justified, i.e., $r_i \in rng(S'.\mathsf{jf} \cap (S'.\mathsf{jf}^?\ ;\ S'.\mathsf{hb}))$. Therefore $\langle r_{i+1}, w\rangle \in S'.\mathsf{cf}_{\mathtt{imm}}\ ;\ [rng(\mathsf{jf} \cap (\mathsf{jf}^?\ ;\ \mathsf{hb}))]\ ;\ S'.\mathsf{po}$.

□

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.

$$x := r_1 \,;\, r_2 := y \quad \rightsquigarrow \quad r_2 := y \,;\, x := r_1 \qquad \text{store/load}$$
$$x := r_1 \,;\, y := r_2 \quad \rightsquigarrow \quad y := r_2 \,;\, x := r_1 \qquad \text{store/store}$$
$$r_1 := x \,;\, r_2 := y \quad \rightsquigarrow \quad r_2 := y \,;\, r_1 := x \qquad \text{load/load}$$
$$r_1 := x \,;\, y := r_2 \quad \rightsquigarrow \quad y := r_2 \,;\, r_1 := x \qquad \text{load/store}$$

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

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 $\mathcal{I}(P_{\mathtt{src}}, P_{\mathtt{tgt}}, G_{\mathtt{tgt}}, S_{\mathtt{src}}, S_{\mathtt{tgt}}, X_{\mathtt{src}}, \mu)$ between the source and target programs $P_{\mathtt{src}}$, $P_{\mathtt{tgt}}$; target execution graphs $G_{\mathtt{tgt}}$; source and target Weakestmo consistent event structures $S_{\mathtt{src}}$, $S_{\mathtt{tgt}}$; justified configuration of the source event structure $X_{\mathtt{src}}$, and function $\mu : S_{\mathtt{src}}.\mathsf{E} \to S_{\mathtt{tgt}}.\mathsf{E}$ that maps events of the source event structure to the events of the target event structure[8].

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 Start). *Let* $P_{\mathtt{src}}$ *and* $P_{\mathtt{tgt}}$ *be source and target programs, s.t.* $P_{\mathtt{src}} \xhookrightarrow{\mathtt{reord}(t,\mathtt{a},\mathtt{b})} P_{\mathtt{tgt}}$*, and let* $G_{\mathtt{tgt}}$ *be* Weakestmo *consistent execution graph, corresponding to the target program* $P_{\mathtt{tgt}}$*. Then* $\mathcal{I}(P_{\mathtt{src}}, P_{\mathtt{tgt}}, G_{\mathtt{tgt}}, S_{\mathrm{init}}(P_{\mathtt{src}}), S_{\mathrm{init}}(P_{\mathtt{tgt}}), \{e_0^s\}, \mu_0)$ *holds where*

- $S_{\mathrm{init}}(P_{\mathtt{src}})$ *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 $\mathcal{M} \subseteq S_{\mathtt{src}}.\mathsf{E} \times S_{\mathtt{tgt}}.\mathsf{E}$ that connects events of the source and target event structure, we use function $\mu : S_{\mathtt{src}}.\mathsf{E} \to S_{\mathtt{tgt}}.\mathsf{E}$. We then use notations $\lceil\!\lceil \cdot \rceil\!\rceil$ and $\lfloor\!\lfloor \cdot \rfloor\!\rfloor$, similarly as in §F.2, to lift function μ to subsets of events and component relations (such as po, jf, co) of the two event structures.

- $S_{\text{init}}(P_{\text{tgt}})$ *is initial target event structure containing only the initialization event* $\{e_0^t\}$;
- μ_0 *is the initial event mapping, s.t.* $\mu_0(e_0^s) = e_0^t$ *and* $\mu_0(e) = \bot$ *for all* $e \neq e_0^s$.

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

Lemma G.4 (Simulation End). *If* $\mathcal{I}(P_{\text{src}}, P_{\text{tgt}}, G_{\text{tgt}}, S_{\text{src}}, S_{\text{tgt}}, X_{\text{src}}, \mu)$ *and* $S_{\text{tgt}} \rhd G_{\text{tgt}}$ *hold, then the execution graph associated with* X_{src}, *denoted as* G_{src}, *has the same behavior as* G_{tgt}, *i.e.,* $\mathcal{B}(G_{\text{src}}) = \mathcal{B}(G_{\text{tgt}})$.

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 $\mathcal{I}$.

The simulation relation $\mathcal{I}$ distinguishes three subsets of events of the target event structure S_{tgt}. Giving that $P_{\text{src}} \xleftarrow{\text{reord}(t,\text{a},\text{b})} P_{\text{tgt}}$ let

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

The simulation $\mathcal{I}$ relation establishes the following properties

(1) For each non-reordered event in S_{tgt} there exists a corresponding event in S_{src}:
  (a) $S_{\text{tgt}}.\text{E} \cap C \subseteq \lceil S_{\text{src}}.\text{E} \rceil$
(2) Labels of events in S_{src} match labels of events in S_{tgt}:
  (a) $\forall e \in S_{\text{src}}.\text{E}.\ S_{\text{src}}.\texttt{lab}(e) = S_{\text{tgt}}.\texttt{lab}(\lceil e \rceil)$
(3) Program order in S_{tgt} between events from A and C corresponds to program order in S_{src}:
  (a) $[A \cup C] ; S_{\text{tgt}}.\text{po} ; [A \cup C] = [A \cup C] ; \lceil S_{\text{src}}.\text{po} \rceil ; [A \cup C]$
  Similarly, program order in S_{tgt} between events from B and C corresponds to program order in S_{src}:
  (b) $[B \cup C] ; S_{\text{tgt}}.\text{po} ; [B \cup C] = [B \cup C] ; \lceil S_{\text{src}}.\text{po} \rceil ; [B \cup C]$
  For immediate program order in S_{src} between events from A and B the following is true:
  (c) $[A] ; \lceil S_{\text{src}}.\text{po}_{\text{imm}} \rceil ; [B] \subseteq S_{\text{tgt}}.\text{ew}^? ; S_{\text{tgt}}.\text{po}_{\text{imm}}{}^{-1} ; S_{\text{tgt}}.\text{cf}_{\text{imm}}{}^?$
  Immediate po predecessor of an event from B in S_{tgt} corresponds to immediate po predecessor of an event from A in S_{tgt}:
  (d) $[C] ; S_{\text{tgt}}.\text{po}_{\text{imm}} ; [B] = [C] ; \lceil S_{\text{src}}.\text{po}_{\text{imm}} ; [\lfloor A \rfloor] ; S_{\text{src}}.\text{po}_{\text{imm}} \rceil ; [B]$
(4) Identity relation in S_{tgt} corresponds to identity or conflict relation in S_{src}:
  - $\lfloor \texttt{id} \rfloor \subseteq S_{\text{src}}.\text{cf}^?$
(5) Justified-from relation in S_{tgt} from an event belonging to A or C to an event belonging to C corresponds to justified-from relation in S_{src}:
  (a) $[A \cup C] ; S_{\text{tgt}}.\text{jf} ; [C] = [A \cup C] ; \lceil S_{\text{src}}.\text{jf} \rceil ; [C]$
  Justified-from relation in S_{tgt} from an event belonging to B or C to an event belonging to B or C corresponds to justified-from relation in S_{src}:
  (b) $[B \cup C] ; S_{\text{tgt}}.\text{jf} ; [B \cup C] = [B \cup C] ; \lceil S_{\text{src}}.\text{jf} \rceil ; [B \cup C]$
  For each justification of an event from A in S_{tgt} there exists corresponding justification in S_{src}:
  (c) $S_{\text{tgt}}.\text{jf} ; [A] \subseteq \lceil S_{\text{src}}.\text{jf} \rceil$
  Each justification of an event from A in S_{src} is either local (i.e., from $\text{jf}^? ; \text{hb}$ preceding write) or the same as in S_{tgt}:

(d) $S_{\mathtt{src}}.\mathtt{jf} \,;\, [\!|A|\!] \subseteq S_{\mathtt{src}}.\mathtt{jf}^? \,;\, S_{\mathtt{src}}.\mathtt{hb} \cup |\!|S_{\mathtt{tgt}}.\mathtt{jf}|\!|$

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

(a) $[B] \,;\, \lceil S_{\mathtt{src}}.\mathtt{ew} \rceil \,;\, [B] \subseteq S_{\mathtt{tgt}}.\mathtt{ew}^?$

Equivalent-writes relation in $S_{\mathtt{tgt}}$ between write events from C corresponds to Equivalent-writes relation in $S_{\mathtt{src}}$:

(b) $[C] \,;\, S_{\mathtt{tgt}}.\mathtt{ew} \,;\, [C] = [C] \,;\, \lceil S_{\mathtt{src}}.\mathtt{ew} \rceil \,;\, [C]$

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

(a) $S_{\mathtt{tgt}}.\mathtt{co} = \lceil S_{\mathtt{src}}.\mathtt{co} \rceil$

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

(a) $G_{\mathtt{src}}.\mathsf{E} = G_{\mathtt{tgt}}.\mathsf{E} \cap S_{\mathtt{tgt}}.\mathsf{E}$

(b) $\forall e \in G_{\mathtt{src}}.\mathsf{E}.\, G_{\mathtt{src}}.\mathtt{lab}(e) = G_{\mathtt{tgt}}.\mathtt{lab}(\lceil e \rceil)$

(c) $\lceil G_{\mathtt{src}}.\mathsf{po} \rceil = G_{\mathtt{tgt}}.\mathsf{po}|_{S_{\mathtt{tgt}}.\mathsf{E}} \setminus \{(b,a)\} \cup \{(a,b)\}$

(d) $\lceil G_{\mathtt{src}}.\mathtt{rf} \rceil = G_{\mathtt{tgt}}.\mathtt{rf}|_{S_{\mathtt{tgt}}.\mathsf{E}}$

(e) $\lceil G_{\mathtt{src}}.\mathtt{co} \rceil = G_{\mathtt{tgt}}.\mathtt{co}|_{S_{\mathtt{tgt}}.\mathsf{E}}$

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

$$(\mathtt{jf} \setminus (\mathtt{jf}^? \,;\, \mathtt{hb})) \,;\, \mathsf{po} \,;\, \mathtt{ew} \subseteq \mathtt{jf} \,;\, (\mathsf{po} \cup \mathtt{lbpat})$$

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

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

Store/Load: Consider the cases.

(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^t is added.

- $e_t \in B$

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

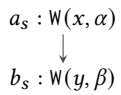
$$S_{\mathtt{src}} \xrightarrow{a_s}{}^{?} S''_{\mathtt{src}} \xrightarrow{b_s} S'_{\mathtt{src}}$$

  Note that it is indeed possible to perform the first step, because instructions a and b are independent (thus the written value α of the event $a_t : \mathtt{W}(x,\alpha)$ cannot depend on the value of the previous event $b_t : \mathtt{R}(y,\beta)$), and because it is always possible to add a write event to

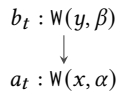
the event structure (because write events do not need to be justified via jf relation). Finally, note that if event b_t depends on some promises in $S'_{\mathtt{tgt}}$, then in $S'_{\mathtt{src}}$ it should depend on the same or smaller set of promises. Indeed, suppose that in $S'_{\mathtt{tgt}}$ event b_t depends via jo path on event a'_t, s.t. $\langle a'_t, a_t\rangle \in S'_{\mathtt{tgt}}.\mathtt{ew}$. In $S'_{\mathtt{src}}$ 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 $S'_{\mathtt{tgt}}$ 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 $S_{\mathtt{src}}$ should already has a matching event a_s. Thus $S'_{\mathtt{src}} = S_{\mathtt{src}}$.
- $e_t \in C$
  In this case construction of the source event structure can simulate the step by adding event e_s with the same label $S_{\mathtt{src}} \xrightarrow{e_s} S'_{\mathtt{src}}$. Since $S'_{\mathtt{tgt}}$ satisfies NO-BAIT-AND-SWITCH then so $S'_{\mathtt{src}}$, because if e belongs to some certification branch in $S'_{\mathtt{tgt}}$ then in $S'_{\mathtt{src}}$ 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.



(a) Source event structure



(b) Target event structure

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 $e_t = b_t : \mathtt{W}(y, \beta)$. To simulate this step, events $a_s : \mathtt{W}(x, \alpha)$ and $b_s : \mathtt{W}(y, \beta)$ are added to the source event structure:

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

  The events a_s and b_s in $S'_{\mathtt{src}}$ depend on the same set of promises and external writes as b_t in $S'_{\mathtt{tgt}}$. Therefore, each of these promise will be certified by the same certification writes as in target event structure $S'_{\mathtt{tgt}}$ 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 $S_{\mathtt{src}}$ should already has a matching event a_s. Thus $S'_{\mathtt{src}} = S_{\mathtt{src}}$.
- $e_t \in C$
  In this case construction of the source event structure can simulate the step by adding event e_s with the same label $S_{\mathtt{src}} \xrightarrow{e_s} S'_{\mathtt{src}}$. Since $S'_{\mathtt{tgt}}$ satisfies NO-BAIT-AND-SWITCH then so $S'_{\mathtt{src}}$.

Load/Load: Consider the cases.

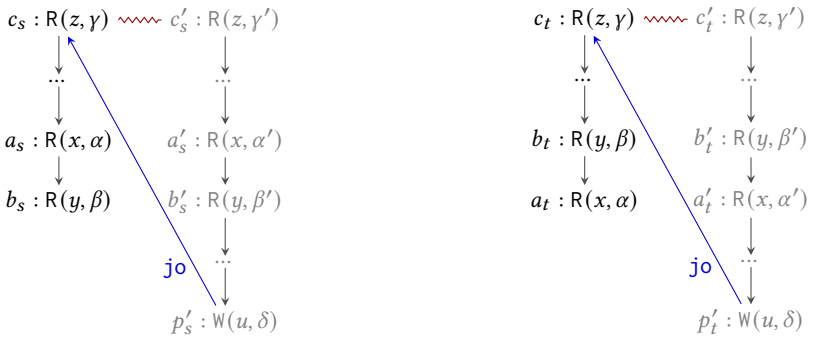
- $e_t \in B$
  Let $e_t = b_t : \mathtt{R}(y, \beta)$ and let $a_t :: \mathtt{R}(x, \alpha)$ be the subsequent event. To simulate this step, events

$a_s : \mathsf{R}(x, \alpha)$ and $b_s : \mathsf{R}(y, \beta)$ are added to the source event structure:

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

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 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.



(a) Source event structure (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 predeccessors 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'_t 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 NO-BAIT-AND-SWITCH it has to be that either these reads are local, i.e., they read from $\mathtt{jf}^?$; 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 $S''_{\mathsf{src}} = S_{\mathsf{src}}$. 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 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'_t 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

(a) Source event structure (b) Target event structure

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 NO-BAIT-AND-SWITCH.

(a) Source event structure (b) Target event structure

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 $S_{\mathtt{src}}$ should already has a matching event a_s. Thus $S'_{\mathtt{src}} = S_{\mathtt{src}}$.
- $e_t \in C$
  In this case construction of the source event structure can simulate the step by adding event e_s with the same label $S_{\mathtt{src}} \xrightarrow{e_s} S'_{\mathtt{src}}$. Since $S'_{\mathtt{tgt}}$ satisfies NO-BAIT-AND-SWITCH then so $S'_{\mathtt{src}}$.

Load/Store: Consider the cases.

- $e_t \in B$
  Let $e_t = b_t : \mathtt{W}(y, \beta)$ and let $a_t : \mathtt{R}(x, \alpha)$ 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 $a_s : \mathtt{R}(x, \alpha)$ and $b_s : \mathtt{W}(y, \beta)$:

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

  Note that event a_s is justified by the same write in $S'_{\mathtt{src}}$ as a_t in $S'_{\mathtt{tgt}}$. 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 $S'_{\mathtt{tgt}}$. Moreover, the set of external writes of these promises remain the same. Therefore, NO-BAIT-AND-SWITCH holds.

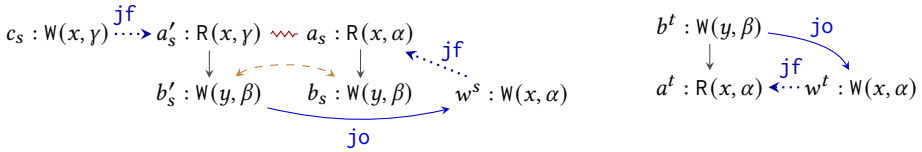
(a) Source event structure (b) Target event structure

Fig. 13. A fragment of event structure construction that justifies load/store reordering. Case when a_t does not depend on b_t via jo path.

– 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 $a'_s : \mathtt{R}(x, \gamma)$ and $b'_s : \mathtt{W}(y, \beta)$:

$$S_{\mathsf{src}} \xrightarrow{a'_s} S''_{\mathsf{src}} \xrightarrow{b'_s} S'_{\mathsf{src}}$$

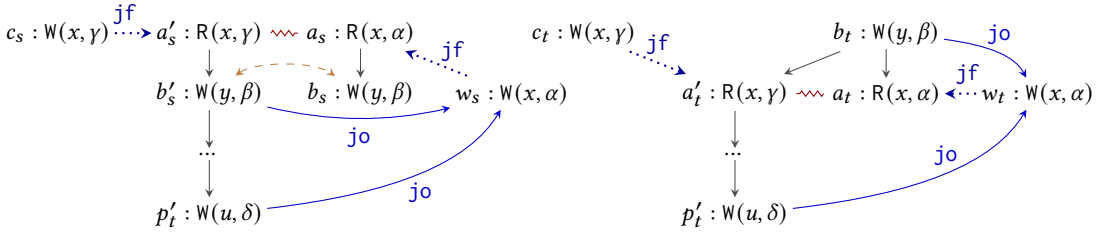
As a justification write for a'_s the construction chooses event $c_s : \mathtt{W}(x, \gamma)$, that is co maximal non-conflicting write from $\mathtt{jf}^?$; hb prefix of a'_s. Since a'_s and b'_s do not depend on promises, the condition NO-BAIT-AND-SWITCH is satisfied trivially.



(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 $\mathtt{jf}^?$; hb preceding write $c_t : \mathtt{W}(x, \gamma)$. We can conclude that S_{src} should contain events $a'_s : \mathtt{R}(x, \gamma)$ and $b'_s : \mathtt{W}(y, \beta)$, s.t. $\mu(a'_s) = a'_t$ and $\mu(b'_s) = b_t$. Therefore $S'_{\mathsf{src}} = S_{\mathsf{src}}$.



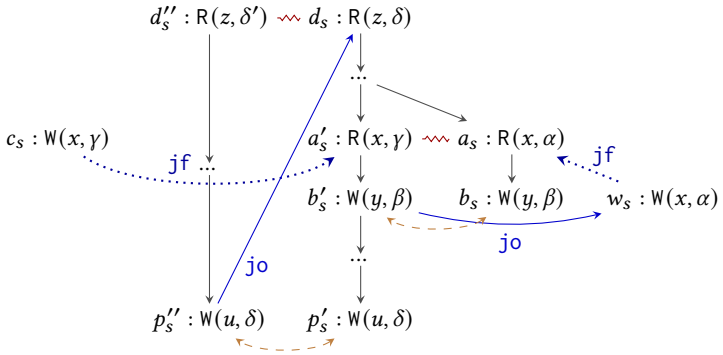
(a) Source event structure (b) Target event structure

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.

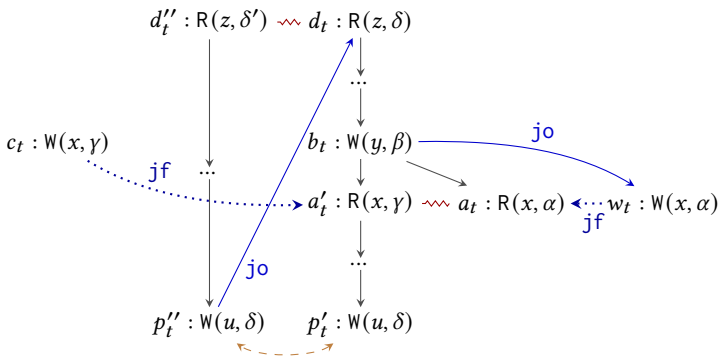
– Suppose that a_t depend on b_t via $\texttt{jo}$ path, and also that a_t, b_t depend on the same set of promises. This case is depicted on Fig. 16. Consider write w_t that justifies a_t. Note that w_t is external write, i.e., $\langle w_t, a_t\rangle \notin \texttt{jf}^?$; $\texttt{hb}$, or otherwise we would get $\mathsf{po} \cup \texttt{jf}$ cycle $b_t \xrightarrow{\texttt{jo}} w_t \xrightarrow{\texttt{jo}} b_t$. Also note that it cannot be the case that $\langle w_t, p_t''\rangle \in \texttt{jf}^?$; $\texttt{hb}$, as required by NO-BAIT-AND-SWITCH, because it also would imply $\mathsf{po} \cup \texttt{jf}$ cycle $w_t \xrightarrow{\texttt{jo}} p_t'' \xrightarrow{\texttt{jo}} w_t$. 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 $a_s' \to b_s' \to \ldots \to p_s'$:

$$S_{\mathsf{src}} \to^* S'_{\mathsf{src}}$$

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



(a) Source event structure



(b) Target event structure

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 $e_t = a_t : \mathtt{R}(x, \alpha)$. We need to consider the same cases as above.

- Consult Fig. 13 again. It can be seen that $S_{\mathtt{src}}$ should already contain suitable event a_s. Therefore $S'_{\mathtt{src}} = S_{\mathtt{src}}$.
- Consult Fig. 14 again. In this case the source event structure is updated by addition of events $a_s : \mathtt{R}(x, \alpha)$ and $b_s : \mathtt{W}(y, \alpha)$:

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

  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 $\mathtt{jf}^?$; $\mathtt{hb}$ prior write.
- Consult Fig. 15 again. In this case the source event structure is updated by addition of events $a_s : \mathtt{R}(x, \alpha)$ and $b_s : \mathtt{W}(y, \alpha)$:

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

  These events form a new certification branch mimicking the branch starting at a_t in $S'_{\mathtt{tgt}}$. All promises on which this branch depends will be certified by the same certification writes as in $S'_{\mathtt{tgt}}$. Additionally, in $S'_{\mathtt{src}}$ this branch also depends on the promise b'_s. This promise is certified by b_s. 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 $a_s : \mathtt{R}(x, \alpha)$ and $b_s : \mathtt{W}(y, \alpha)$:

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

  Similarly to the previous case, these events form a new certification branch mimicking the branch starting at a_t in $S'_{\mathtt{tgt}}$. 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 $S_{\mathtt{src}} \xrightarrow{e_s} S'_{\mathtt{src}}$. Since $S'_{\mathtt{tgt}}$ satisfies NO-BAIT-AND-SWITCH then so $S'_{\mathtt{src}}$.

G.2 Elimination of Redundant Accesses

Next we prove soundness of eliminations.

Suppose that thread t of a program P contains two consecutive instructions $\mathtt{a}$; $\mathtt{b}$ such that these instructions access the same location and one of the instructions is redundant. For example, $\mathtt{a}$ and $\mathtt{b}$ 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 $\mathtt{elim}(t, \mathtt{a}, \mathtt{b})$ removes or replaces redundant instruction, and leaves other threads unchanged.

We consider three kind of eliminations: store/load, store/store, load/load[9]:

$$x := r_1 \,;\, r_2 := x \quad \rightsquigarrow \quad x := r_1 \,;\, r_2 := r_1 \qquad \text{store/load}$$

$$x := r_1 \,;\, x := r_2 \quad \rightsquigarrow \quad x := r_2 \qquad \text{store/store}$$

$$r_1 := x \,;\, r_2 := x \quad \rightsquigarrow \quad r_1 := x \,;\, r_2 := r_1 \qquad \text{load/load}$$

Again, we assume that all accesses are relaxed.

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

[9]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 $\mathcal{I}$ 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 $\mathcal{I}(P_{\mathtt{src}}, P_{\mathtt{tgt}}, G_{\mathtt{tgt}}, S_{\mathtt{src}}, S_{\mathtt{tgt}}, X_{\mathtt{src}}, \mu)$ establish a connection between the source and target programs $P_{\mathtt{src}}$, $P_{\mathtt{tgt}}$; target execution graphs $G_{\mathtt{tgt}}$; source and target Weakestmo consistent event structures $S_{\mathtt{src}}$, $S_{\mathtt{tgt}}$; justified configuration of the source event structure $X_{\mathtt{src}}$, and function $\mu : S_{\mathtt{src}}.\mathsf{E} \to S_{\mathtt{tgt}}.\mathsf{E}$ that maps events of the source event structure to the events of the target event structure.

The simulation relation $\mathcal{I}$ distinguishes three subsets of events of the target event structure $S_{\mathtt{tgt}}$. Giving that $P_{\mathtt{src}} \xhookrightarrow{\mathtt{elim}(t,\mathtt{a},\mathtt{b})} P_{\mathtt{tgt}}$ let

- $A \subseteq S_{\mathtt{tgt}}.\mathsf{E}$ be a subset of events obtained as a result of executing instruction a;
- $B \subseteq S_{\mathtt{tgt}}.\mathsf{E}$ be a subset of events obtained as a result of executing instruction b;
- $C \triangleq S_{\mathtt{tgt}}.\mathsf{E} \setminus (A \cup B)$ 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 $\mathcal{I}$ relation establishes the following properties

(1) For each event in $S_{\mathtt{tgt}}$ there exists a corresponding event in in $S_{\mathtt{src}}$:
  (a) $S_{\mathtt{tgt}}.\mathsf{E} \subseteq \lceil S_{\mathtt{src}}.\mathsf{E} \rceil$
  Function μ is injective on non-eliminated events:
  (b) $\forall e_1, e_2 \in S_{\mathtt{src}}.\mathsf{E}.\, \mu(e_1) = \mu(e_2) \in C \implies e_1 = e_2$
(2) Label of at least one of the events forming a redundant access pair in $S_{\mathtt{src}}$ matches the label of its image in $S_{\mathtt{tgt}}$:
  (a) $\forall \langle e_a, e_b \rangle \in [\lfloor A \cup B \rfloor] ; S_{\mathtt{src}}.\mathsf{po}_{\mathtt{imm}} ; [\lfloor A \cup B \rfloor]$ either
    - $A \neq \emptyset$ and $S_{\mathtt{src}}.\mathtt{lab}(e_a) = S_{\mathtt{tgt}}.\mathtt{lab}(\lceil e_a \rceil)$, or
    - $B \neq \emptyset$ and $S_{\mathtt{src}}.\mathtt{lab}(e_b) = S_{\mathtt{tgt}}.\mathtt{lab}(\lceil e_b \rceil)$

  Labels of events from C match in both event structures:
  (b) $\forall e \in \lfloor C \rfloor.\ S_{\mathtt{src}}.\mathtt{lab}(e) = S_{\mathtt{tgt}}.\mathtt{lab}(\lceil e \rceil)$
(3) Program order in $S_{\mathtt{tgt}}$ between events from A and C corresponds to program order in $S_{\mathtt{src}}$:
  (a) $[A \cup C] ; S_{\mathtt{tgt}}.\mathsf{po} ; [A \cup C] = [A \cup C] ; \lceil S_{\mathtt{src}}.\mathsf{po} \rceil ; [A \cup C]$
  Similarly, program order in $S_{\mathtt{tgt}}$ between events from B and C corresponds to program order in $S_{\mathtt{src}}$:
  (b) $[B \cup C] ; S_{\mathtt{tgt}}.\mathsf{po} ; [B \cup C] = [B \cup C] ; \lceil S_{\mathtt{src}}.\mathsf{po} \rceil ; [B \cup C]$
  Pair of redundant accesses in $S_{\mathtt{src}}$ maps to the same event in $S_{\mathtt{tgt}}$:
  (c) $[A \cup B] ; \lceil S_{\mathtt{src}}.\mathsf{po}_{\mathtt{imm}} \rceil ; [A \cup B] \subseteq \mathtt{id}$
(4) Justified-from edge in $S_{\mathtt{src}}$ to an event belonging to A or B corresponds either to the same event in $S_{\mathtt{tgt}}$ or to a justified-from edge in $S_{\mathtt{tgt}}$:
  (a) $\lceil S_{\mathtt{src}}.\mathtt{jf} \rceil ; [A \cup B] \subseteq S_{\mathtt{tgt}}.\mathtt{jf}^?$
  Justified-from edge in $S_{\mathtt{tgt}}$ to an event belonging to A or B corresponds either to a justified-from edge in $S_{\mathtt{src}}$:

(b) $S_{\mathsf{tgt}}.\mathtt{jf} \,;\, [A \cup B] \subseteq [\![S_{\mathsf{src}}.\mathtt{jf}]\!]$
Justified-from relation in S_{tgt} to an event belonging to C corresponds to justified-from relation in S_{src}:
(c) $S_{\mathsf{tgt}}.\mathtt{jf} \,;\, [C] = [\![S_{\mathsf{src}}.\mathtt{jf}]\!] \,;\, [C]$
(5) Equivalent-writes relation in S_{tgt} corresponds to Equivalent-writes relation in S_{src}:
(a) $S_{\mathsf{tgt}}.\mathtt{ew} = [\![S_{\mathsf{src}}.\mathtt{ew}]\!]$
(6) Coherence relation in S_{tgt} corresponds to the coherence relation in S_{src}:
(a) $S_{\mathsf{tgt}}.\mathtt{co} = [\![S_{\mathsf{src}}.\mathtt{co}]\!]$
(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 $a \in A \cap G_{\mathsf{tgt}}.\mathsf{E}$ or $b \in B \cap G_{\mathsf{tgt}}.\mathsf{E}$
(a) $[\![G_{\mathsf{src}}.\mathsf{E}]\!] = G_{\mathsf{tgt}}.\mathsf{E} \cap S_{\mathsf{tgt}}.\mathsf{E}$
Note that equations for lab, 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.

$$(\mathtt{jf} \setminus (\mathtt{jf}^? \,;\, \mathtt{hb})) \,;\, \mathsf{po} \,;\, \mathtt{ew} \subseteq \mathtt{jf} \,;\, (\mathsf{po} \cup \mathtt{lbpat})$$

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

(a) Source event structure (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 $e_t = a_t : \mathtt{W}(x, \alpha)$. To simulate this step, the construction adds events $a_s : \mathtt{W}(x, \alpha)$ and $b_s : \mathtt{R}(x, \alpha)$ to the source event structure:
  $$S_{\mathsf{src}} \xrightarrow{a_s} S''_{\mathsf{src}} \xrightarrow{b_s} S'_{\mathsf{src}}$$
  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 $\mathsf{po} \subseteq \mathtt{jf}^? \,;\, \mathtt{hb}$ 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 $S_{\mathsf{src}} \xrightarrow{e_s} S'_{\mathsf{src}}$. Since S'_{tgt} satisfies NO-BAIT-AND-SWITCH then so S'_{src}.

Store/Load: Consider the cases.

(a) Source event structure (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 $e_t = b_t : \mathtt{W}(x, \beta)$. To simulate this step, the construction adds events $a_s : \mathtt{W}(x, \alpha)$ and $b_s : \mathtt{R}(x, \beta)$ to the source event structure:
  $$S_{\mathtt{src}} \xrightarrow{a_s} S''_{\mathtt{src}} \xrightarrow{b_s} S'_{\mathtt{src}}$$
  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 $S'_{\mathtt{src}}$, they will be certified similarly as promises of b_t in $S'_{\mathtt{tgt}}$, 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 $S_{\mathtt{src}} \xrightarrow{e_s} S'_{\mathtt{src}}$. Since $S'_{\mathtt{tgt}}$ satisfies NO-BAIT-AND-SWITCH then so $S'_{\mathtt{src}}$.

Load/Load: Consider the cases.

(a) Source event structure (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 $e_t = a_t : \mathtt{R}(x, \alpha)$. Let $c_t : \mathtt{W}(x, \alpha)$ be a write event chosen to justify a_t. To simulate this step, the construction adds events $a_s : \mathtt{R}(x, \alpha)$ and $b_s : \mathtt{R}(x, \alpha)$ to the source event structure:
  $$S_{\mathtt{src}} \xrightarrow{a_s} S''_{\mathtt{src}} \xrightarrow{b_s} S'_{\mathtt{src}}$$
  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 $\mathtt{jf}$; $\mathsf{po} \subseteq \mathtt{jf}^?$; $\mathtt{hb}$ 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 $S'_{\mathtt{src}}$, they will be certified similarly as promises of a_t in $S'_{\mathtt{tgt}}$, 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 $S_{\mathtt{src}} \xrightarrow{e_s} S'_{\mathtt{src}}$. Since $S'_{\mathtt{tgt}}$ satisfies NO-BAIT-AND-SWITCH then so $S'_{\mathtt{src}}$.