Reconciling Event Structures with Modern Multiprocessors

Evgenii Moiseenko
St. Petersburg University, Russia
JetBrains Research, Russia
e.moiseenko@2012.spbu.ru

Anton Podkopaev
National Research University Higher School of Economics, Russia
MPI-SWS, Germany
JetBrains Research, Russia
anton.podkopaev@jetbrains.com

Ori Lahav
Tel Aviv University, Israel
orilahav@tau.ac.il

Orestis Melkonian
University of Edinburgh, UK
melkon.or@gmail.com

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

Abstract

Weakestmo is a recently proposed memory consistency model that uses event structures to resolve the infamous "out-of-thin-air" problem and to enable efficient compilation to hardware. Nevertheless, this latter property—compilation correctness—has not yet been formally established.

This paper closes this gap by establishing correctness of the intended compilation schemes from Weakestmo to a wide range of formal hardware memory models (x86, POWER, ARMv7, ARMv8) in the Coq proof assistant. Our proof is the first that establishes correctness of compilation of an event-structure-based model that forbids "out-of-thin-air" behaviors, as well as the first mechanized compilation proof of a weak memory model supporting sequentially consistent accesses to such a range of hardware platforms. Our compilation proof goes via the recent Intermediate Memory Model (IMM), which we suitably extend with sequentially consistent accesses.

2012 ACM Subject Classification Theory of computation → Logic and verification; Software and its engineering → Concurrent programming languages

Keywords and phrases Weak Memory Consistency, Event Structures, IMM, Weakestmo.

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

A major research problem in concurrency semantics is to develop a weak memory model that allows load-to-store reordering (a.k.a. *load buffering*, LB) combined with compiler optimizations (*e.g.,* elimination of fake dependencies), while forbidding "out-of-thin-air" behaviors [18, 11, 5, 14]. This problem can be illustrated with the following two programs accessing locations x and y that are initialized to 0. The annotated outcome $a = b = 1$ ought to be allowed for LB-fake (because $1 + a * 0$ can be optimized to 1 and then the instructions of thread 1 executed out of order) and forbidden for LB-data (where no optimizations are applicable).

$$\begin{array}{l} a := [x] \quad /\!/ 1 \\ [y] := 1 + a * 0 \end{array} \;\Big\|\; \begin{array}{l} b := [y] \quad /\!/ 1 \\ [x] := b \end{array} \qquad \text{(LB-fake)}$$

$$\begin{array}{l} a := [x] \quad /\!/ 1 \\ [y] := a \end{array} \;\Big\|\; \begin{array}{l} b := [y] \quad /\!/ 1 \\ [x] := b \end{array} \qquad \text{(LB-data)}$$

Among the proposed models that correctly distinguish between these two programs is the recent Weakestmo model [6]. Weakestmo was developed in response to certain limitations of earlier models, such as the "promising semantics" of Kang *et al.* [12], namely that (*i*) they did not cover the whole range of C/C++ concurrency features and that (*ii*) they did not support the intended compilation schemes to hardware.

Being flexible in its design, Weakestmo addresses the former point. It supports all usual features of the C/C++11 model [3] and can easily be adapted to support any new concurrency features that may be added in the future. It does not, however, provide an adequate answer to the latter point. Because of the difficulty of establishing correctness of the intended compilation schemes to hardware architectures that permit load-store reordering (*i.e.,* POWER, ARMv7, ARMv8), Chakraborty and Vafeiadis [6] only establish correctness of suboptimal schemes that add (unnecessary) explicit fences to prevent load-store reordering.

In this paper, we address this major limitation of the Weakestmo paper. We establish in Coq correctness of the intended compilation schemes to a wide range of hardware architectures that includes the major ones: x86-TSO [17], POWER [1], ARMv7 [1], ARMv8 [21]. The compilation schemes, whose correctness we prove, do not require any fences or fake dependencies for relaxed accesses. Because of a technical limitation of our setup (see §6), however, compilation of read-modify-write (RMW) accesses to ARMv8 uses a load-reserve/store-conditional loop (similar to that of ARMv7 and POWER) as opposed to the newly introduced ARMv8 instructions for certain kinds of RMWs.

The main challenge in this proof is to reconcile the different ways in which hardware models and Weakestmo allow load-store reordering. Unlike most models at the programming language level, hardware models (such as ARMv8) do not execute instructions in sequence; they instead keep track of dependencies between instructions and ensure that no dependency cycles ever arise in a single execution. In contrast, Weakestmo executes instructions in order, but simultaneously considers multiple executions to justify an execution where a load reads a value that indirectly depends upon a later store. Technically, these multiple executions together form an *event structure*, upon which Weakestmo places various constraints.

The high-level proof structure is shown in Fig. 1. We reuse IMM, an *intermediate memory model*, introduced by Podkopaev *et al.* [19] as an abstraction over all major existing hardware memory models. To support Weakestmo compilation, we extend IMM with *sequentially consistent* (SC) accesses following the RC11 model [14]. As IMM is very much a hardware-like model (*e.g.,* it tracks dependencies), the main result is compilation from Weakestmo to IMM (indicated by the bold arrow). The other arrows in the figure are extensions of previous results to account for SC accesses, while double arrows indicate results for two compilation schemes.

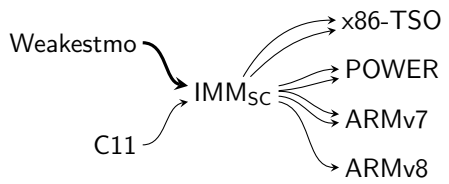


Figure 1 Results proved in this paper.

The complexity of the proof is also evident from the size of the Coq development. We have written about 30K lines of Coq definitions and proof scripts on top of an existing infrastructure of about another 20K lines (defining IMM, the aforementioned hardware models and many lemmas about them). As part of developing the proof, we also had to mechanize the Weakestmo definition in Coq and to fix some minor deficiencies in the original definition, which were revealed by our proof effort.

(a) $G_{\mathtt{LB}}$: Execution graph of LB. **(b)** Execution of LB-data and LB-fake.

Figure 2 Executions of LB and LB-data/LB-fake with outcome $a = b = 1$.

To the best of our knowledge, our proof is the first proof of correctness of compilation of an event-structure-based memory model. It is also the first mechanized compilation proof of a weak memory model supporting sequentially consistent accesses to such a range of hardware architectures. The latter, although fairly straightforward in our case, has had a history of wrong compilation correctness arguments (see [14] for details).

Outline We start with an informal overview of IMM, Weakestmo, and our compilation proof (§2). We then present a fragment of Weakestmo formally (§3) and its compilation proof (§4). Subsequently, we extend these results to cover SC accesses (§5), discuss related work (§6) and conclude (§7). The associated proof scripts can be found in the supplementary material.

2 Overview of the Compilation Correctness Proof

To get an idea about the IMM and Weakestmo memory models, consider a version of the LB-fake and LB-data programs from §1 with no dependency in thread 1:

$$\begin{array}{l|l} a := [x] \quad \mathit{// 1} & b := [y] \quad \mathit{// 1} \\ [y] := 1 & [x] := b \end{array} \tag{LB}$$

As we will see, the annotated outcome is allowed by both IMM and Weakestmo, albeit in different ways. The different treatment of load-store reordering affects the outcomes of other programs. For example, IMM forbids the annotate outcome of LB-fake by treating it exactly as LB-data, whereas Weakestmo allows the outcome by treating LB-fake exactly as LB.

2.1 An Informal Introduction to IMM

IMM is a *declarative* (also called *axiomatic*) model identifying a program's semantics with a set of *execution graphs*, or just *executions*. As an example, Fig. 2a contains $G_{\mathtt{LB}}$, an IMM execution graph of LB corresponding to an execution yielding the annotated behavior.

Vertices of execution graphs, called *events*, represent memory accesses either due to the initialization of memory or to the execution of program instructions. Each event is labeled with the type of the access (*e.g.,* R for reads, W for writes), the location accessed, and the value read or written. Memory initialization consists of a set of events labeled $\mathtt{W}(x, 0)$ for each location x used in the program; for conciseness, however, we depict the initialization events as a single event with label Init.

Edges of execution graphs represent different relations on events. In Fig. 2, three different relations are depicted. The *program order* relation (po) totally orders events originated from the same thread according to their order in the program, as well as the initialization event(s) before all other events. The *reads-from* relation (rf) relates a write event to the read events that read from it. Finally, the *preserved program order* (ppo) is a subset of the program

order relating events that cannot be executed out of order. Such ppo edges arise whenever there is a dependency chain between the corresponding instructions (*e.g.,* a write storing the value read by a prior read).

Because of the syntactic nature of ppo, IMM conflates the executions of LB-data and LB-fake leading to the outcome $a = b = 1$ (see Fig. 2b). This choice is in line with hardware memory models; it means, however, that IMM is not suitable as a memory model for a programming language (because, as argued in §1, LB-fake can be transformed to LB by an optimizing compiler).

The executions of a program are constructed in two steps.[1] First, a thread-local semantics determines the sequential executions of each thread, where the values returned by each read access are chosen non-deterministically (among the set of *all* possible values), and the executions of different threads are combined into a single execution. Then, the execution graphs are filtered by a *consistency predicate*, which determines which executions are allowed (i.e., are IMM-consistent). These IMM-consistent executions form the program's semantics.

IMM-consistency checks three basic constraints:

Completeness: Every read event reads from precisely one write with the same location and value;

Coherence: For each location x, there is a total ordering of x-related events extending the program order so that each read of x reads from the most recent prior write according to that total order; and

Acyclic dependency: There is no cycle consisting only of ppo and rf edges.

The final constraint disallows executions in which an event recursively depends upon itself, as this pattern can lead to "out-of-thin-air" outcomes. Specifically, the execution in Fig. 2b, which represents the annotated behavior of LB-fake and LB-data, is *not* IMM-consistent because of the $(\mathtt{ppo} \cup \mathtt{rf})$-cycle. In contrast, $G_{\mathtt{LB}}$ is IMM-consistent.

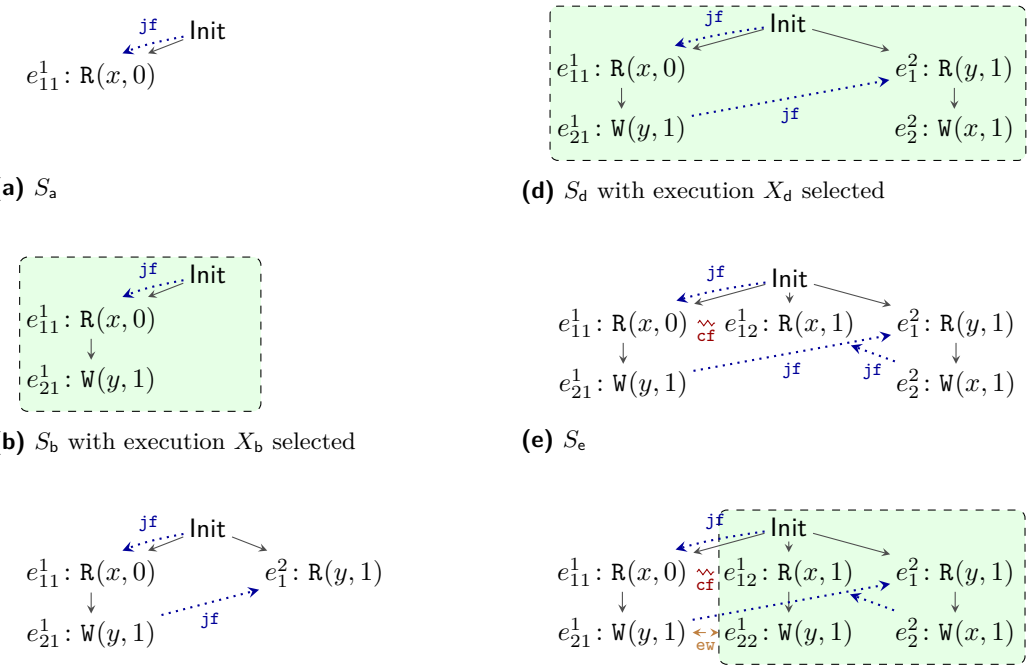
2.2 An Informal Introduction to Weakestmo

We move on to Weakestmo, which also defines the program's semantics as a set of execution graphs. However, they are constructed differently—extracted from a final *event structure*, which Weakestmo incrementally builds for a program.

An event structure represents multiple executions of a programs in a single graph. Like execution graphs, event structures contain a set of events and several relations among them. Like execution graphs, the *program order* (po) orders events according to each thread's control flow. However, unlike execution graphs, po is not necessarily total among the events of a given thread. Events of the same thread that are not po-ordered are said to be in *conflict* (cf) with one another, and cannot belong to the same execution. Such conflict events arise when two read events originate from the same read instruction (*e.g.,* representing executions where the reads return different values). Moreover, cf "extends downwards": events that depend upon conflicting events (*i.e.,* have conflicting po-predecessors) are also in conflict with one other. In pictures, we typically show only the *immediate conflict* edges (between reads originating from the same instruction) and omit the conflict edges between events po-after immediately conflicting ones.

Event structures are constructed incrementally starting from an event structure consisting only of the initialization events. Then, events corresponding to the execution of program

[1] For a detailed formal description of the graphs and their construction process we refer the reader to [19, §2.2].



(a) S_a

(b) S_b with execution X_b selected

(c) S_c

(d) S_d with execution X_d selected

(e) S_e

(f) S_f with execution X_f selected

Figure 3 A run of Weakestmo witnessing the annotated outcome of LB.

instructions are added one at a time. We start by executing the first instruction of a program's thread. Then, we may execute the second instruction of the same thread or the first instruction of another thread, and so on.

As an example, Fig. 3 constructs an event structure for LB. Fig. 3a depicts the event structure S_a obtained from the initial event structure by executing $a := [x]$ in LB's thread 1. As a result of the instruction execution, a read event e^1_{11} : R$(x, 0)$ is added.

Whenever the event added is a read, Weakestmo has to justify the returned value from an appropriate write event. In this case, there is only one write to x—the initialization write—and so S_a has a *justified from* edge, denoted jf, going to e^1_{11} in S_a. This is a requirement of Weakestmo: each read event in an event structure has to be justified from exactly one write event with the same value and location. (This requirement is analogous to the *completeness* requirement in IMM-consistency for execution graphs.) Since events are added in program order and read events are always justified from existing events in the event structure, po $\cup$ jf is guaranteed to be acyclic by construction.

The next three steps (Figures 3b to 3d) simply add a new event to the event structure. Notice that unlike IMM executions, Weakestmo event structures do not track syntactic dependencies, *e.g.,* S_d in Fig. 3d does not contain a ppo edge between e^2_1 and e^2_2. This is precisely what allows Weakestmo to assign the same behavior to LB and LB-fake: they have exactly the same event structures. As a programming-language-level memory model, Weakestmo supports optimizations removing fake dependencies.

The next step (Fig. 3e) is more interesting because it showcases the key distinction between event structures and execution graphs, namely that event structures may contain more than one execution for each thread. Specifically, the transition from S_d to S_e reruns the first instruction of thread 1 and adds a new event e^1_{12} justified from a different write event. We say that this new event *conflicts* (cf) with e^1_{11} because they cannot both occur

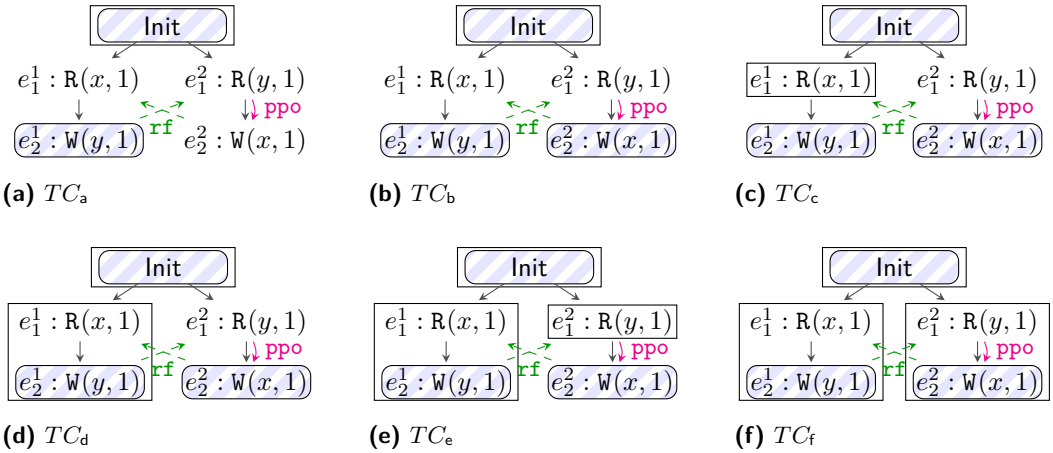


Figure 4 Traversal configurations for $G_{\mathtt{LB}}$.

in a single execution. Because of conflicts, po in event structures does not totally order all events of a thread; *e.g.,* e_{11}^1 and e_{12}^1 are not po-ordered in S_{e}. Two events of the same thread are conflicted precisely when they are not po-ordered.

The final construction step (Fig. 3f) demonstrates another Weakestmo feature. Conflicting write events writing the same value to the same location (*e.g.,* e_{21}^1 and e_{22}^1 in S_{f}) may be declared *equal writes*, *i.e.,* connected by an equivalence relation ew.[2]

The ew relation is used to define Weakestmo's version of the reads-from relation, rf, which relates a read to all (non-conflicted) writes *equal* to the write justifying the read. For example, e_1^2 reads from both e_{21}^1 and e_{22}^1.

The Weakestmo's rf relation is used for extraction of program executions. An execution graph G is *extracted* from an event structure S denoted $S \rhd G$ if G is a maximal conflict-free subset of S, it contains only *visible* events (to be defined in §3), and every read event in G reads from some write in G according to $S.\mathtt{rf}$. Two execution graphs can be extracted from S_{f}: $\{\mathsf{Init}, e_{11}^1, e_{21}^1, e_1^2, e_2^2\}$ and $\{\mathsf{Init}, e_{12}^1, e_{22}^1, e_1^2, e_2^2\}$ representing the outcomes $a = 0 \wedge b = 1$ and $a = b = 1$ respectively.

2.3 Weakestmo to IMM Compilation: High-Level Proof Structure

In this paper, we assume that Weakestmo is defined for the same assembly language as IMM (see [19, Fig. 2]) extended with SC accesses and refer to this language as L. Having that, we show the correctness of *the identity* mapping as a compilation scheme from Weakestmo to IMM in the following theorem.

▶ **Theorem 1.** *Let prog be a program in* L*, and G be an* IMM*-consistent execution graph of prog. Then there exists an event structure S of prog under* Weakestmo *such that $S \rhd G$.*

To prove the theorem, we must show that Weakestmo may construct the needed event structure in a step by step fashion. If the IMM-consistent execution graph G contains no $\mathsf{po} \cup \mathtt{rf}$ cycles, then the construction is completely straightforward: G itself is a Weakestmo-consistent event structure (setting jf to be just rf), and its events can be added in any order extending $\mathsf{po} \cup \mathtt{rf}$.

[2] In this paper, we take ew to be reflexive, whereas it is is irreflexive in Chakraborty and Vafeiadis [6]. Our ew is the reflexive closure of the one in [6].

The construction becomes tricky for IMM-consistent execution graphs, such as $G_{\mathtt{LB}}$, that contain po∪rf cycles. Due to the cycle(s), G cannot be directly constructed as a (conflict-free) Weakestmo event structure. We must instead construct a larger event structure S containing multiple executions, one of which will be the desired graph G. Roughly, for each $\mathtt{po} \cup \mathtt{rf}$ cycle in G, we have to construct an immediate conflict in the event structure.

To generate the event structure S, we rely on a basic property of IMM-consistent execution graphs shown by Podkopaev *et al.* [19, §§6,7], namely that execution graphs can be *traversed* in a certain order, *i.e.,* its events can be *issued* and *covered* in that order, so that in the end all events are covered. The traversal captures a possible execution order of the program that yields the given execution. In that execution order, events are not added according to program order, but rather according to *preserved program order* (ppo) in two steps. Events are first issued when all their dependencies have been resolved, and are later covered when all their po-prior events have been covered.

In more detail, a traversal of an IMM-consistent execution graph G is a sequence of traversal steps between *traversal configurations.* A traversal configuration TC of an execution graph G is a pair of sets of events, $\langle C, I\rangle$, called the *covered* and *issued* set respectively. As an example, Fig. 4 presents all six (except for the initial one) traversal configurations of the execution graph $G_{\mathtt{LB}}$ of LB from Fig. 2a, with the issued set marked by ◯ and the covered set marked by □.

A traversal might be seen as an execution of an abstract machine which is allowed to perform write instructions out-of-order but has to execute everything else in order. The first option corresponds to issuing a write event, and the second option to covering an event. The traversal strategy has certain constraints. To issue a write event, all external reads that it depends upon must read from issued events, while to cover an event, all its po-predecessors must also be covered.[3] For example, a traversal cannot issue $e_2^2\colon \mathtt{W}(x,1)$ before issuing $e_2^1\colon \mathtt{W}(y,1)$ in Fig. 4, or cover $e_1^1\colon \mathtt{R}(x,1)$ before issuing $e_2^2\colon \mathtt{W}(x,1)$.

According to Podkopaev *et al.* [19, Prop. 6.5], every IMM-consistent execution graph G has a full traversal of the following form:

$$G \vdash TC_{\text{init}}(G) \longrightarrow TC_1 \longrightarrow TC_2 \longrightarrow \ldots \longrightarrow TC_{\text{final}}(G)$$

where the initial configuration, $TC_{\text{init}}(G) \triangleq \langle G.\mathsf{Init}, G.\mathsf{Init}\rangle$, has covered/issued only G's initial events and the final configuration, $TC_{\text{final}}(G) \triangleq \langle G.\mathtt{E}, G.\mathtt{W}\rangle$, has covered all G's events and issued all its write events.

We then construct the event structure S following a full traversal of G. We define a simulation relation, $\mathcal{I}(prog, G, TC, S, X)$, between the program *prog*, the current traversal configuration TC of execution G and the current event structure's state $\langle S, X\rangle$, where X is a subset of events corresponding to a particular execution graph extracted from the event structure S.

Our simulation proof is divided into the following three lemmas.

▶ **Lemma 2** (Simulation Start). *Let prog be a program of* L*, and G be an* IMM*-consistent execution graph of prog. Then $\mathcal{I}(prog, G, TC_{\text{init}}(G), S_{\text{init}}(prog), S_{\text{init}}(prog).\mathtt{E})$ holds.*

▶ **Lemma 3** (Weak Simulation Step). *If $\mathcal{I}(prog, G, TC, S, X)$ and $G \vdash TC \longrightarrow TC'$ hold, then there exist S' and X' such that $\mathcal{I}(prog, G, TC', S', X')$ and $S \rightarrow^* S'$ hold.*

[3] For readers familiar with PS [12], issuing a write event corresponds to promising a message, and covering an event to normal execution of an instruction.

▶ **Lemma 4** (Simulation End). *If $\mathcal{I}(prog, G, TC_{\text{final}}(G), S, X)$ holds, then the execution graph associated with X is isomorphic to G.*

The proof of Theorem 1 then proceeds by induction on the length of the traversal $G \vdash TC_{\text{init}}(G) \longrightarrow^* TC_{\text{final}}(G)$. Lemma 2 serves as the base case, Lemma 3 is the induction step simulating each traversal step with a number of event structure construction steps, and Lemma 4 concludes the proof.

The proofs of Lemmas 2 and 4 are technical but fairly straightforward. (We define $\mathcal{I}$ in a way that makes these lemmas immediate.) In contrast, Lemma 3 is much more difficult to prove. As we will see, simulating a traversal step sometimes requires us to construct a new branch in the event structure, *i.e.,* to add multiple events (see Section 4.3).

2.4 Weakestmo to IMM Compilation Correctness by Example

Before presenting any formal definitions, we conclude this overview section by showcasing the construction used in the proof of Lemma 3 on execution graph $G_{\mathtt{LB}}$ in Fig. 2a following the traversal of Fig. 4. We have actually already seen the sequence of event structures constructed in Fig. 3. Note that, even though Figures 3 and 4 have the same number of steps, there is no one-to-one correspondence between them as we explain below.

Consider the last event structure S_{f} from Fig. 3. A subset of its events X_{f} marked by , which we call a *simulated execution*, is a maximal conflict-free subset of S_{f} and all read events in X_{f} read from some write in X_{f} (*i.e.,* are justified from a write deemed "equal" to some write in X_{f}). Then, by definition, X_{f} is extracted from S_{f}. Also, an execution graph induced by X_{f} is isomorphic to $G_{\mathtt{LB}}$. That is, construction of S_{f} for LB shows that in Weakestmo it is possible to observe the same behavior as $G_{\mathtt{LB}}$. Now, we explain how we construct S_{f} and choose X_{f}.

During the simulation, we maintain the relation $\mathcal{I}(prog, G, TC, S, X)$ connecting a program $prog$, its execution graph G, its traversal configuration TC, an event structure S, and a subset of its events X. Among other properties (presented in Section 4.2), the relation states that all issued and covered events of TC have exact counterparts in X, and that X can be extracted from S.

The initial event structure and X_{Init} consist of only initial events. Then, following issuing of event $e_2^1 \colon \mathtt{W}(y, 1)$ in TC_{a} (see Fig. 4a), we need to add a branch to the event structure that has $\mathtt{W}(y, 1)$ in it. Since Weakestmo requires adding events according to program order, we first need to add a read event corresponding to '$a := [x]$' of LB's thread 1. Each read event in an event structure has to be justified from somewhere. In this case, the only write event to location x is the initial one. That is, the added read event e_{11}^1 is justified from it (see Fig. 3a). In the general case, having more than one option, we would choose a 'safe' write event for an added read event to be justified from, *i.e.,* the one which the corresponding branch is 'aware' of already and being justified from which would not break consistency of the event structure. After that, a write event $e_{21}^1 \colon \mathtt{W}(y, 1)$ can be added po-after e_{11}^1 (see Fig. 3b), and $\mathcal{I}(\mathtt{LB}, G_{\mathtt{LB}}, TC_{\mathsf{a}}, S_{\mathsf{b}}, X_{\mathsf{b}})$ holds for $X_{\mathsf{b}} = \{\mathsf{Init}, e_{11}^1, e_{21}^1\}$.

Next, we need to simulate the second traversal step (see Fig. 4b), which issues $\mathtt{W}(x, 1)$. As with the previous step, we first need to add a read event related to the first read instruction of LB's thread 2 (see Fig. 3c). However, unlike the previous step, the added event e_1^2 has to get value 1, since there is a dependency between instructions in thread 2. As we mentioned earlier, the traversal strategy guarantees that $e_2^1 \colon \mathtt{W}(y, 1)$ is issued at the moment of issuing $e_2^2 \colon \mathtt{W}(x, 1)$, so there is the corresponding event in the event structure to justify the read event e_1^2 from. Now, the write event $e_2^2 \colon \mathtt{W}(y, 1)$ representing e_2^2 can be added to the event

structure (see Fig. 3d) and $\mathcal{I}(\mathrm{LB}, G_{\mathtt{LB}}, TC_{\mathsf{b}}, S_{\mathsf{d}}, X_{\mathsf{d}})$ holds for $X_{\mathsf{d}} = \{\mathsf{Init}, e^1_{11}, e^1_{21}, e^2_1, e^2_2\}$.

In the third traversal step (see Fig. 4c), the read event $e^1_1 \colon \mathtt{R}(x, 1)$ is covered. To have a representative event for e^1_1 in the event structure, we add e^1_{12} (see Fig. 3e). It is justified from e^2_2, which writes the needed value 1. Also, e^1_{12} represents an alternative to e^1_{11} execution of the first instruction of thread 1, so the events are in conflict.

However, we cannot choose a simulated execution X related to TC_{c} and S_{e} by the simulation relation since X has to contain e^1_{12} and a representative for $e^1_2 \colon \mathtt{W}(y, 1)$ (in S_{e} it is represented by e^1_{21}) while being conflict-free. Thus, the event structure has to make one other step (see Fig. 3f) and add the new event e^1_{22} to represent $e^1_2 \colon \mathtt{W}(y, 1)$. Now, the simulated execution contains everything needed, $X_{\mathsf{f}} = \{\mathsf{Init}, e^1_{12}, e^1_{22}, e^2_1, e^2_2\}$.

Since X_{f} has to be extracted from S_{f}, every read event in X has to be connected via an $\mathtt{rf}$ edge to an event in X.[4] To preserve the requirement, we connect the newly added event e^1_{22} and e^1_{21} via an $\mathtt{ew}$ edge, *i.e.,* marking them to be equal writes.[5] This induces an $\mathtt{rf}$ edge between e^1_{22} and e^2_1. That is, $\mathcal{I}(\mathrm{LB}, G_{\mathtt{LB}}, TC_{\mathsf{c}}, S_{\mathsf{f}}, X_{\mathsf{f}})$ holds.

To simulate the remaining traversal steps (Figures 4d to 4f), we do not need to modify S_{f} because it already contains counterparts for the newly covered events and, moreover, the execution graph associated with X_{f} is isomorphic to $G_{\mathtt{LB}}$. That is, we just need to show that $\mathcal{I}(\mathrm{LB}, G_{\mathtt{LB}}, TC_{\mathsf{d}}, S_{\mathsf{f}}, X_{\mathsf{f}})$, $\mathcal{I}(\mathrm{LB}, G_{\mathtt{LB}}, TC_{\mathsf{e}}, S_{\mathsf{f}}, X_{\mathsf{f}})$, and $\mathcal{I}(\mathrm{LB}, G_{\mathtt{LB}}, TC_{\mathsf{f}}, S_{\mathsf{f}}, X_{\mathsf{f}})$ hold.

3 Formal Definition of Weakestmo

In this section, we introduce the notation used in the rest of the paper and define the Weakestmo memory model. For simplicity, we present only a minimal fragment of Weakestmo containing only relaxed reads and writes. For the definition of the full Weakestmo model, we refer the readers to Chakraborty and Vafeiadis [6] and to our Coq development.

Notation Given relations R_1 and R_2, we write $R_1 \,;\, R_2$ for their sequential composition. Given relation R, we write $R^?$, R^+ and R^* to denote its reflexive, transitive and reflexive-transitive closures. We write $\mathtt{id}$ to denote the identity relation (*i.e.,* $\mathtt{id} \triangleq \{\langle x, x\rangle\}$). For a set A, we write $[A]$ to denote the identity relation restricted to A (that is, $[A] \triangleq \{\langle a, a\rangle \mid a \in A\}$). Hence, for instance, we may write $[A] \,;\, R \,;\, [B]$ instead of $R \cap (A \times B)$. We also write $[e]$ to denote $[\{e\}]$ if e is not a set.

Given a function $f \colon A \to B$, we denote by $=_f$ the set of f-equivalent elements: ($=_f \triangleq \{\langle a, b\rangle \in A \times A \mid f(a) = f(b)\}$). In addition, given a relation R, we denote by $R|_{=f}$ the restriction of R to f-equivalent elements ($R|_{=f} \triangleq R \cap =_f$), and by $R|_{\neq f}$ be the restriction of R to non-f-equivalent elements ($R|_{\neq f} \triangleq R \setminus =_f$).

3.1 Events, Threads and Labels

Events, $e \in \mathsf{Event}$, and *thread identifiers*, $t \in \mathsf{Tid}$, are represented by natural numbers. We treat the thread with identifier 0 as the *initialization* thread. We let $x \in \mathsf{Loc}$ to range over *locations*, and $v \in \mathsf{Val}$ over *values*.

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

[4] Actually, it is easy to show that there could be only one such event since equal writes are in conflict and X is conflict-free.

[5] Note that we could have left e^1_{22} without any outgoing $\mathtt{ew}$ edges since the choice of equal writes for newly added events in Weakestmo is non-deterministic. However, that would not preserve the simulation relation.

- $\mathtt{R}(x, v)$ — a read of value v from location x.
- $\mathtt{W}(x, v)$ — a write of value v to location x.

Given a label l the functions $\mathtt{typ}$, $\mathtt{loc}$, $\mathtt{val}$ return (when applicable) its type (*i.e.,* $\mathtt{R}$ or $\mathtt{W}$), location and value correspondingly. When a specific function assigning labels to events is clear from the context, we abuse the notations $\mathtt{R}$ and $\mathtt{W}$ to denote the sets of all events labelled with the corresponding type. We also use subscripts to further restrict this set to a specific location (*e.g.,* $\mathtt{W}_x$ denotes the set of write events operating on location x.)

3.2 Event Structures

An *event structure* S is a tuple $\langle \mathtt{E}, \mathtt{tid}, \mathtt{lab}, \mathtt{po}, \mathtt{jf}, \mathtt{ew}, \mathtt{co} \rangle$ where:

- $\mathtt{E}$ is a set of events, *i.e.,* $\mathtt{E} \subseteq \mathsf{Event}$.
- $\mathtt{tid} : \mathtt{E} \to \mathsf{Tid}$ is a function assigning a thread identifier to every event. We treat events with the thread identifier equal to 0 as *initialization events* and denote them as Init, that is $\mathsf{Init} \triangleq \{e \in \mathtt{E} \mid \mathtt{tid}(e) = 0\}$.
- $\mathtt{lab} : \mathtt{E} \to \mathsf{Lab}$ is a function assigning a label to every event in $\mathtt{E}$.
- $\mathtt{po} \subseteq \mathtt{E} \times \mathtt{E}$ is a strict partial order on events, called *program order*, that tracks their precedence in the control flow of the program. Initialization events are $\mathtt{po}$-before all other events, whereas non-initialization events can only be $\mathtt{po}$-before events from the same thread.

  Not all events of a thread are necessarily ordered by $\mathtt{po}$. We call such $\mathtt{po}$-unordered non-initialization events of the same thread *conflicting* events. The corresponding binary relation $\mathtt{cf}$ is defined as follows:

$$\mathtt{cf} \triangleq ([\mathtt{E} \setminus \mathsf{Init}] \,;\, =_{\mathtt{tid}} \,;\, [\mathtt{E} \setminus \mathsf{Init}]) \setminus (\mathtt{po} \cup \mathtt{po}^{-1})^{?}$$

- $\mathtt{jf} \subseteq [\mathtt{E} \cap \mathtt{W}] \,;\, (=_{\mathtt{loc}} \cap =_{\mathtt{val}}) \,;\, [\mathtt{E} \cap \mathtt{R}]$ is the *justified from* relation, which relates a write event to the reads it justifies. We require that reads are not justified by conflicting writes (*i.e.,* $\mathtt{jf} \cap \mathtt{cf} = \emptyset$) and $\mathtt{jf}^{-1}$ be *functional* (*i.e.,* whenever $\langle w_1, r \rangle, \langle w_2, r \rangle \in \mathtt{jf}$, then $w_1 = w_2$). We also define the notion of *external* justification: $\mathtt{jfe} \triangleq \mathtt{jf} \setminus \mathtt{po}$. A read event is externally justified from a write if the write is not $\mathtt{po}$-before the read.
- $\mathtt{ew} \subseteq [\mathtt{E} \cap \mathtt{W}] \,;\, (\mathtt{cf} \cap =_{\mathtt{loc}} \cap =_{\mathtt{val}})^{?} \,;\, [\mathtt{E} \cap \mathtt{W}]$ is an equivalence relation called the *equal-writes* relation. Equal writes have the same location and value, and (unless identical) are in conflict with one another.
- $\mathtt{co} \subseteq [\mathtt{E} \cap \mathtt{W}] \,;\, (=_{\mathtt{loc}} \setminus \mathtt{ew}) \,;\, [\mathtt{E} \cap \mathtt{W}]$ is the *coherence* order, a strict partial order that relates non-equal write events with the same location. We require that coherence be closed with respect to equal writes (*i.e.,* $\mathtt{ew} \,;\, \mathtt{co} \,;\, \mathtt{ew} \subseteq \mathtt{co}$) and total with respect to $\mathtt{ew}$ on writes to the same location:

$$\forall x \in \mathsf{Loc}.\ \forall w_1, w_2 \in \mathtt{W}_x.\ \langle w_1, w_2 \rangle \in \mathtt{ew} \cup \mathtt{co} \cup \mathtt{co}^{-1}$$

Given an event structure S, we use "dot notation" to refer to its components (*e.g.,* $S.\mathtt{E}$, $S.\mathtt{po}$). For a set A of events, we write $S.A$ for the set $A \cap S.\mathtt{E}$ (for instance, $S.\mathtt{W}_x = \{e \in S.\mathtt{E} \mid \mathtt{typ}(S.\mathtt{lab}(e)) = \mathtt{W} \wedge \mathtt{loc}(S.\mathtt{lab}(e)) = x\}$). Further, for $e \in S.\mathtt{E}$, we write $S.\mathtt{typ}(e)$ to retrieve $\mathtt{typ}(S.\mathtt{lab}(e))$. Similar notation is used for the functions $\mathtt{loc}$ and $\mathtt{val}$. Given a set of thread identifiers T, we write $S.\mathtt{thread}(T)$ to denote the set of events belonging to one of the threads in T, *i.e.,* $S.\mathtt{thread}(T) \triangleq \{e \in S.\mathtt{E} \mid S.\mathtt{tid}(e) \in T\}$. When $T = \{\mathtt{thread}(t)\}$ is a singleton, we often write $S.\mathtt{thread}(t)$ instead of $S.\mathtt{thread}(\{t\})$.

We define the immediate $\mathtt{po}$ and $\mathtt{cf}$ edges of an event structure as follows:

$$S.\mathtt{po}_{\mathtt{imm}} \triangleq S.\mathtt{po} \setminus (S.\mathtt{po} \,;\, S.\mathtt{po}) \qquad S.\mathtt{cf}_{\mathtt{imm}} \triangleq S.\mathtt{cf} \cap (S.\mathtt{po}_{\mathtt{imm}}{}^{-1} \,;\, S.\mathtt{po}_{\mathtt{imm}})$$

An event e_1 is an immediate po-predecessor of e_2 if e_1 is po-before e_2 and there is no event po-between them. Two conflicting events are immediately conflicting if they have the same immediate po-predecessor.[6]

3.3 Event Structure Construction

Given a program *prog*, we construct its event structures operationally in a way that guarantees completeness (*i.e.,* that every read is justified from some write) and $\mathtt{po} \cup \mathtt{jf}$ acyclicity. We start with an event structure containing only the initialization events and add one event at a time following each thread's semantics.

For the thread semantics, we assume reductions of the form $\sigma \xrightarrow{e} \sigma'$ between thread states $\sigma, \sigma' \in \mathtt{ThreadState}$ and labeled by the event $e \in \mathtt{E}$ generated by that execution step. Given a thread t and a sequence of events $e_1, \dots, e_n \in S.\mathtt{thread}(t)$ in immediate po succession (*i.e.,* $\langle e_i, e_{i+1} \rangle \in S.\mathtt{po}_{\mathtt{imm}}$ for $1 \leq i < n$) starting from a first event of thread t (*i.e.,* $dom(S.\mathtt{po}; [e_1]) \subseteq \mathsf{Init}$), we can add an event e po-after that sequence of events provided that there exist thread states $\sigma_1, \dots, \sigma_n$ and σ' such that $prog(t) \xrightarrow{e_1} \sigma_1 \xrightarrow{e_2} \sigma_2 \cdots \xrightarrow{e_n} \sigma_n \xrightarrow{e} \sigma'$, where $prog(t)$ is the initial thread state of thread t of the program *prog*. By construction, this means that the newly added event e will be in conflict with all other events of thread t besides $e_1, \dots, e_n$.

Further, when the new event e is a read event, it has to be justified from an existing write event, so as to ensure completeness and prevent "out-of-thin-air" values. The write event is picked non-deterministically from all non-conflicting writes with the same location as the new read event. Similarly, when e is a write event, its position in co order should be chosen. It can be done by either picking an ew equivalence class and including the new write in it, or by putting the new write immediately after some existing write in co order. At each step, we also check for *event structure consistency* (to be defined in Def. 5): If the event structure obtained after the addition of the new event is inconsistent, it is discarded.

3.4 Event Structure Consistency

To define consistency, we first need a number of auxiliary definitions. The *happens-before* order $S.\mathtt{hb}$ is a generalization of the program order. Besides the program order edges, it includes certain *synchronization* edges (captured by the *synchronizes with* relation, $S.\mathtt{sw}$).

$$S.\mathtt{hb} \triangleq (S.\mathtt{po} \cup S.\mathtt{sw})^+$$

For the fragment covered in this section, there are no synchronization edges (*i.e.,* $\mathtt{sw} = \emptyset$), and so hb and po coincide. In the full model,[7] however, certain justification edges (*e.g.,* between release/acquire accesses) contribute to sw and hence to hb.

The *extended conflict* relation $S.\mathtt{ecf}$ extends the notion of conflicting events to account for hb; two events are in extended conflict if they happen after conflicting events.

$$S.\mathtt{ecf} \triangleq (S.\mathtt{hb}^{-1})^? \,;\, S.\mathtt{cf} \,;\, S.\mathtt{hb}^?$$

As already mentioned in §2, the *reads-from* relation, $S.\mathtt{rf}$, of a Weakestmo event structure is derived. It is defined as an extension of $S.\mathtt{jf}$ to all $S.\mathtt{ew}$-equivalent writes.

$$S.\mathtt{rf} \triangleq (S.\mathtt{ew} \,;\, S.\mathtt{jf}) \setminus S.\mathtt{cf}$$

[6] Our definition of immediate conflicts differs from that of [6] and is easier to work with. The two definitions are equivalent if the set of initialization events is non-empty.

[7] The full model is presented in [6] and also in our Coq development.

Note that unlike $S.\mathtt{jf}^{-1}$, the relation $S.\mathtt{rf}^{-1}$ is not functional. This does not cause any problems, however, since all the writes from whence a read reads have the same location and value and are in conflict with one another.

The relation $S.\mathtt{fr}$, called *from-read* or *reads-before*, places read events before subsequent writes.

$$S.\mathtt{fr} \triangleq S.\mathtt{rf}^{-1} ; S.\mathtt{co}$$

The *extended coherence* $S.\mathtt{eco}$ is a strict partial order that orders events operating on the same location. (It is almost total on accesses to a given location, except that it does not order equal writes nor reads reading from the same write.)

$$S.\mathtt{eco} \triangleq (S.\mathtt{co} \cup S.\mathtt{rf} \cup S.\mathtt{fr})^{+}$$

We observe that in our model, $\mathtt{eco}$ is equal to $\mathtt{rf} \cup \mathtt{co};\mathtt{rf}^{?} \cup \mathtt{fr};\mathtt{rf}^{?}$, similar to the corresponding definitions about execution graphs in the literature.[8]

The last ingredient that we need for event structure consistency is the notion of *visible* events, which will be used to constrain external justifications. We define it in a few steps. Let e be some event in S. First, consider all write events used to externally justify e or one of its justification ancestors. The relation $S.\mathtt{jfe} ; (S.\mathrm{po} \cup S.\mathtt{jf})^{*}$ defines this connection formally. Among that set of write events restrict attention to those conflicting with e, and call that set M. That is, $M \triangleq dom(S.\mathtt{cf} \cap (S.\mathtt{jfe} ; (S.\mathrm{po} \cup S.\mathtt{jf})^{*}) ; [e])$. Event e is *visible* if all writes in M have an equal write that is po-related with e. Formally,[9]

$$S.\mathtt{Vis} \triangleq \{e \in S.\mathtt{E} \mid S.\mathtt{cf} \cap (S.\mathtt{jfe} ; (S.\mathrm{po} \cup S.\mathtt{jf})^{*}) ; [e] \subseteq S.\mathtt{ew} ; (S.\mathrm{po} \cup S.\mathrm{po}^{-1})^{?}\}$$

Intuitively, visible events cannot depend on conflicting events: for every such justification dependence, there ought to be an equal non-conflicting write.

Consistency places a number of additional constraints on event structures. First, it checks that there is no redundancy in the event structure: immediate conflicts arise only because of read events justified from non-equal writes. Second, it extends the constraints about $\mathtt{cf}$ to the extended conflict $\mathtt{ecf}$; namely that no event can conflict with itself or be justified from a conflicting event. Third, it checks that reads are justified either from events of the same thread or from visible events of other threads. Finally, it ensures *coherence*, *i.e.,* that executions restricted to accesses on a single location do not have any weak behaviors.

▶ **Definition 5.** *An event structure S is said to be* consistent *if the following conditions hold.*

- $dom(S.\mathtt{cf}_{\mathtt{imm}}) \subseteq S.\mathtt{R}$ ($\mathtt{cf}_{\mathtt{imm}}$-READ)
- $S.\mathtt{jf} ; S.\mathtt{cf}_{\mathtt{imm}} ; S.\mathtt{jf}^{-1} ; S.\mathtt{ew}$ *is irreflexive.* ($\mathtt{cf}_{\mathtt{imm}}$-JUSTIFICATION)
- $S.\mathtt{ecf}$ *is irreflexive.* ($\mathtt{ecf}$-IRREFLEXIVITY)
- $S.\mathtt{jf} \cap S.\mathtt{ecf} = \emptyset$ ($\mathtt{jf}$-NON-CONFLICT)
- $dom(S.\mathtt{jfe}) \subseteq S.\mathtt{Vis}$ ($\mathtt{jfe}$-VISIBLE)
- $S.\mathtt{hb} ; S.\mathtt{eco}^{?}$ *is irreflexive.* (COHERENCE)

[8] This equivalence equivalence does not hold in the original Weakestmo model [6]. To make the equivalence hold, we made $\mathtt{ew}$ transitive, and require $\mathtt{ew} ; \mathtt{co} ; \mathtt{ew} \subseteq \mathtt{co}$.

[9] Note, that in [6] the definition of the visible events is slightly more verbose. We proved in Coq that our simpler definition is equivalent to the one given there.

3.5 Execution Extraction

The last part of Weakestmo is the extraction of executions from an event structure. An execution is essentially a conflict-free event structure.

▶ **Definition 6.** *An* execution graph G *is a tuple* $\langle \mathtt{E}, \mathtt{tid}, \mathtt{lab}, \mathtt{po}, \mathtt{rf}, \mathtt{co} \rangle$ *where its components are defined similarly as in the case of an event structure with the following exceptions:*

- $\mathtt{po}$ *is required to be total on the set of events from the same thread. Thus, execution graphs have no conflicting events,* i.e., $\mathtt{cf} = \emptyset$.
- *The* $\mathtt{rf}$ *relation is given explicitly instead of being derived. Also, there are no* $\mathtt{jf}$ *and* $\mathtt{ew}$ *relations.*
- $\mathtt{co}$ *totally orders write events operating on the same location.*

All derived relations are defined similarly as for event structures. Next we show how to extract an execution graph from the event structure.

▶ **Definition 7.** *A set of events* X *is called* extracted from S *if the following conditions are met:*

- X *is conflict-free,* i.e., $[X] \,;\, S.\mathtt{cf} \,;\, [X] = \emptyset$.
- X *is* $S.\mathtt{rf}$*-complete,* i.e., $X \cap S.\mathtt{R} \subseteq \mathit{codom}([X] \,;\, S.\mathtt{rf})$.
- X *contains only visible events of* S*,* i.e., $X \subseteq S.\mathtt{Vis}$.
- X *is* $\mathtt{hb}$*-downward-closed,* i.e., $\mathit{dom}(S.\mathtt{hb} \,;\, [X]) \subseteq X$.

Given an event structure S and extracted subset of its events X, it is possible to associate with X an execution graph G simply by restricting the corresponding components of S to X:

$$
\begin{array}{lll}
G.\mathtt{E} = X & G.\mathtt{tid} = S.\mathtt{tid}|_X & G.\mathtt{lab} = S.\mathtt{lab}|_X \\
G.\mathtt{po} = [X] \,;\, S.\mathtt{po} \,;\, [X] & G.\mathtt{rf} = [X] \,;\, S.\mathtt{rf} \,;\, [X] & G.\mathtt{co} = [X] \,;\, S.\mathtt{co} \,;\, [X]
\end{array}
$$

We say that such execution graph G is *associated with* X and that it is *extracted* from the event structure: $S \rhd G$.

Weakestmo additionally defines another consistency predicate to further filter out some of the extracted execution graphs. In the Weakestmo fragment we consider, this additional consistency predicate is trivial—every extracted execution satisfies it—and so we do not present it here. In the full model, execution consistency checks atomicity of read-modify-write instructions, and sequential consistency for SC accesses.

4 Compilation Proof for Weakestmo

In this section, we outline our correctness proof for the compilation from Weakestmo to the various hardware models. As already mentioned, our proof utilizes IMM [19]. In the following, we briefly present IMM for the fragment of the model containing only relaxed reads and writes (Section 4.1), our simulation relation (Section 4.2) for the compilation from Weakestmo to IMM, and outline the argument as to why the simulation relation is preserved (Section 4.3). Mapping from IMM to the hardware models has already been proved correct by Podkopaev *et al.* [19], so we do not present this part here. Later, in §5, we will extend the IMM mapping results to cover SC accesses.

As a further motivating example for this section consider yet another variant of the load buffering program shown in Fig. 5. As we will see, its annotated weak behavior is allowed by IMM and also by Weakestmo, albeit in a different way. The argument for constructing the Weakestmo event structure that exhibits the weak behavior from the given IMM execution graph is non-trivial.

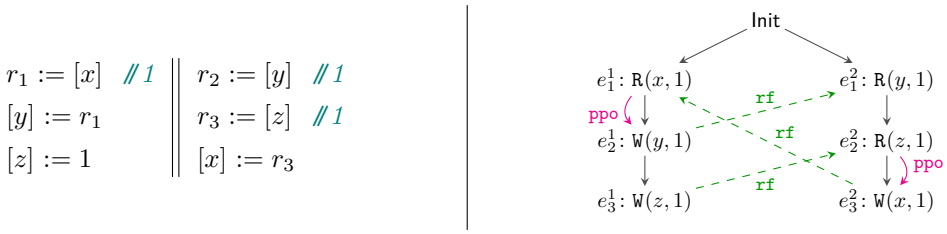


Figure 5 A variant of the load-buffering program (left) and the IMM graph G corresponding to its annotated weak behavior (right).

4.1 The Intermediate Memory Model IMM

In order to discuss the proof, we briefly present a simplified version of the formal IMM definition, where we have omitted constraints about RMW accesses and fences.

▶ **Definition 8.** *An* IMM execution graph G *is an execution graph (Def. 6) extended with one additional component: the* preserved program order $\mathtt{ppo} \subseteq [\mathtt{R}] ; \mathtt{po} ; [\mathtt{W}]$.

Preserved program order edges correspond to syntactic dependencies guaranteed to be preserved by all major hardware platforms. For example, the execution graph in Fig. 5 has two ppo edges corresponding to the data dependencies via registers r_1 and r_3. (The full IMM definition [19] distinguishes between the different types of dependencies—control, data, adress–and includes them as separate components of execution graphs. In the full model, ppo is actually derived from the more basic dependencies.)

IMM-consistency checks completeness, coherence, and acyclicity:[10]

▶ **Definition 9.** *An* IMM *execution graph* G *is* IMM-consistent *if*

- $codom(G.\mathtt{rf}) = G.\mathtt{R}$, (COMPLETENESS)
- $G.\mathtt{hb} ; G.\mathtt{eco}^?$ *is irreflexive, and* (COHERENCE)
- $G.\mathtt{rf} \cup G.\mathtt{ppo}$ *is acyclic.* (NO-THIN-AIR)

As we can see, the execution graph G of Fig. 5 is IMM-consistent because every read of the graph reads from some write event and, moreover, the COHERENCE and NO-THIN-AIR properties hold.

4.2 Simulation Relation for Weakestmo to IMM Proof

In this section, we define the simulation relation $\mathcal{I}$, which is used for the simulation of a traversal of an IMM-consistent execution graph by a Weakestmo event structure presented in Section 2.3.

The way we define $\mathcal{I}(prog, G, \langle C, I\rangle, S, X)$ induces a strong connection between events in the execution graph G and the event structure S. We make this connection explicit with the function $\mathtt{s2g}_{G,S} : S.\mathtt{E} \to G.\mathtt{E}$, which maps events of the event structure S into the events of the execution graph G, such that e and $\mathtt{s2g}_{G,S}(e)$ belong to the same thread and have the same po-position in the thread.[11] Note that $\mathtt{s2g}_{G,S}$ is defined for all events $e \in S.\mathtt{E}$, meaning

[10] Again, this is a simplified presentation for a fragment of the model. We refer the reader to Podkopaev *et al.* [19] or our Coq development for the full definition, which further distinguishes between internal and external rf edges.

[11] Here we assume existence and uniqueness of such a function. In our Coq development, we have a different representation of execution graphs which makes the existence and uniqueness questions trivial.

that the event structure S does not contain any redundant events that do not correspond to events in the IMM execution graph G. The function $\mathtt{s2g}_{G,S}$, however, does not have to be injective: in particular, events e and e' that are in immediate conflict in S have the same $\mathtt{s2g}_{G,S}$-image in G. In the rest of the paper, whenever G and S are clear from the context, we omit the G, S subscript from $\mathtt{s2g}$.

In the context of a function $\mathtt{s2g}$ (for some G and S), we also use $[\![\cdot]\!]$ and $\lfloor\!\lfloor\cdot\rfloor\!\rfloor$ to lift $\mathtt{s2g}$ to sets and relations:

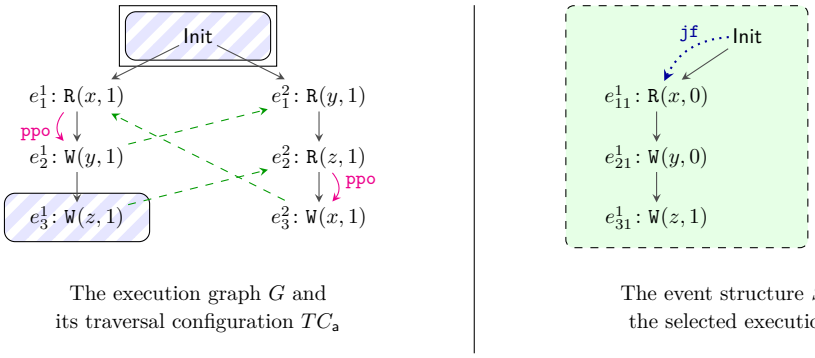
$$\text{for } A_S \subseteq S.\mathtt{E} : [\![A_S]\!] \triangleq \{\mathtt{s2g}(e) \mid e \in A_S\}$$
$$\text{for } A_G \subseteq G.\mathtt{E} : \lfloor\!\lfloor A_G \rfloor\!\rfloor \triangleq \{e \in S.\mathtt{E} \mid \mathtt{s2g}(e) \in A_G\}$$
$$\text{for } R_S \subseteq S.\mathtt{E} \times S.\mathtt{E} : [\![R_S]\!] \triangleq \{\langle \mathtt{s2g}(e), \mathtt{s2g}(e')\rangle \mid \langle e, e'\rangle \in R_S\}$$
$$\text{for } R_G \subseteq G.\mathtt{E} \times G.\mathtt{E} : \lfloor\!\lfloor R_G \rfloor\!\rfloor \triangleq \{\langle e, e'\rangle \in S.\mathtt{E} \times S.\mathtt{E} \mid \langle \mathtt{s2g}(e), \mathtt{s2g}(e')\rangle \in R_G\}$$

For example, $\lfloor\!\lfloor C \rfloor\!\rfloor$ denotes a subset of S's events whose $\mathtt{s2g}$-images are covered events in G, and $[\![S.\mathtt{rf}]\!]$ denotes a relation on events in G whose $\mathtt{s2g}$-preimages in S are related by $S.\mathtt{rf}$.

We define the relation $\mathcal{I}(prog, G, \langle C, I\rangle, S, X)$ to hold if the following conditions are met:

1. G is an IMM-consistent execution of $prog$.
2. S is a Weakestmo-consistent event structure of $prog$.
3. X is an extracted subset of S.
4. S and X corresponds precisely to all covered and issued events and their po-predecessors:
   - $[\![S.\mathtt{E}]\!] = [\![X]\!] = C \cup dom(G.\mathtt{po}^? ; [I])$

   (Note that C is closed under po-predecessors, so $dom(G.\mathtt{po}^? ; [C]) = C$.)
5. Each S event has the same thread, type, modifier, and location as its corresponding G event. In addition, covered and issued events in X have the same value as their corresponding ones in G.
   a. $\forall e \in S.\mathtt{E}.\ S.\{\mathtt{tid}, \mathtt{typ}, \mathtt{loc}, \mathtt{mod}\}(e) = G.\{\mathtt{tid}, \mathtt{typ}, \mathtt{loc}, \mathtt{mod}\}(\mathtt{s2g}(e))$
   b. $\forall e \in X \cap \lfloor\!\lfloor C \cup I \rfloor\!\rfloor.\ S.\mathtt{val}(e) = G.\mathtt{val}(\mathtt{s2g}(e))$
6. Program order in S corresponds to program order in G:
   - $[\![S.\mathtt{po}]\!] \subseteq G.\mathtt{po}$
7. Identity relation in G corresponds to identity or conflict relation in S:
   - $\lfloor\!\lfloor \mathtt{id} \rfloor\!\rfloor \subseteq S.\mathtt{cf}^?$
8. Reads in S are justified by writes that have already been observed by the corresponding events in G. Moreover, covered events in X are justified by a write corresponding to that read from the corresponding read in G:
   a. $[\![S.\mathtt{jf}]\!] \subseteq G.\mathtt{rf}^? ; G.\mathtt{hb}^?$
   b. $[\![S.\mathtt{jf} ; [X \cap \lfloor\!\lfloor C \rfloor\!\rfloor]]\!] \subseteq G.\mathtt{rf}$
9. Every write event justifying some external read event should be $S.\mathtt{ew}$-equal to some issued write event in X:
   - $dom(S.\mathtt{jfe}) \subseteq dom(S.\mathtt{ew} ; [X \cap \lfloor\!\lfloor I \rfloor\!\rfloor])$
10. Equal writes in S correspond to the same write event in G:
    - $[\![S.\mathtt{ew}]\!] \subseteq \mathtt{id}$
11. Every non-trivial $S.\mathtt{ew}$ equivalence class contains an issued write in X:
    - $S.\mathtt{ew} \subseteq (S.\mathtt{ew} ; [X \cap \lfloor\!\lfloor I \rfloor\!\rfloor] ; S.\mathtt{ew})^?$
12. Coherence edges in S correspond to coherence or identity edges in G. (We will explain in Section 4.3 why a coherence edge in S might correspond to an identity edge in G.)
    - $[\![S.\mathtt{co}]\!] \subseteq G.\mathtt{co}^?$



The execution graph G and its traversal configuration TC_{a}

The event structure S_{a} and the selected execution X_{a}

Figure 6 The execution graph G, its traversal configuration TC_{a}, the related event structure S_{a}, and the selected execution X_{a}. Covered events are marked by □ and issued ones by ◯. Events belonging to the selected execution are marked by ◯.

As an example, consider the execution G from Fig. 5, the traversal configuration $TC_{\mathsf{a}} \triangleq \langle\{\mathsf{Init}\}, \{\mathsf{Init}, e_3^1\}\rangle$, and the event structure S_{a} shown in Fig. 6. We will show that $\mathcal{I}(prog, G, TC_{\mathsf{a}}, S_{\mathsf{a}}, X_{\mathsf{a}})$, where $X_{\mathsf{a}} \triangleq S_{\mathsf{a}}.\mathtt{E}$, holds.

Take $\mathsf{s2g}_{G,S_{\mathsf{a}}} = \{\mathsf{Init} \mapsto \mathsf{Init}, e_{11}^1 \mapsto e_1^1, e_{21}^1 \mapsto e_2^1, e_{31}^1 \mapsto e_3^1\}$. Given that $\mathtt{cf} = \mathtt{ew} = \emptyset$, the consistency constraints hold immediately. For example, condition 8 holds because e_{11}^1 is justified by Init, which happens before it. Finally, note that only e_{31}^1 and e_3^1 are required to have the same value by constraint 5, the other related thread events only need to have the same type and address.

The definition of the simulation relation $\mathcal{I}$ renders the proofs of Lemmas 2 and 4 straightforward. Specifically, for Lemma 2, the initial configuration $TC_{\mathrm{init}}(G)$ containing only the initialization events is simulated by the initial event structure S_{init} as all the constraints are trivially satisfied ($S_{\mathrm{init}}.\mathtt{po} = S_{\mathrm{init}}.\mathtt{jf} = S_{\mathrm{init}}.\mathtt{ew} = S_{\mathrm{init}}.\mathtt{co} = \emptyset$).

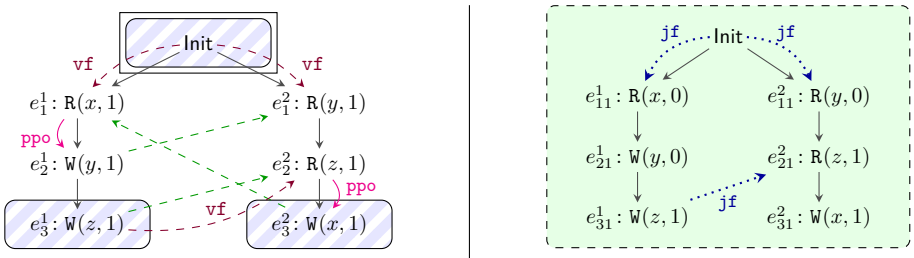
For Lemma 4, since $TC_{\mathrm{final}}(G)$ covers all events of G, property 5 implies that the labels of the events in X are equal to the corresponding events of G; property 6 means that po is the same between them; property 8 means that rf is the same between them; properties 7 and 12 together mean that co is the same. Therefore, G and the execution corresponding to X are isomorphic.

4.3 Simulation Step Proof Outline

We next outline the proof of Lemma 3, which states that the simulation relation $\mathcal{I}$ can be restored after a traversal step.

Suppose that $\mathcal{I}(prog, G, TC, S, X)$ holds for some $prog$, G, TC, S, and X, and we need to simulate a traversal step $TC \longrightarrow TC'$ that either covers or issues an event of thread t. Then we need to produce an event structure S' and a subset of its events X' such that $\mathcal{I}(prog, G, TC', S', X')$ holds. Whenever thread t has any uncovered issued write events, Weakestmo might need to take multiple steps from S to S' so as to add any missing events po-before the uncovered issued writes of thread t. Borrowing the terminology of the "promising semantics" [12], we refer to these steps as constructing a certification branch for the issued write(s).

Before we present the construction, let us return to the example of Fig. 5. Consider the traversal step from configuration TC_{a} to to configuration $TC_b \triangleq \langle\{\mathsf{Init}\}, \{\mathsf{Init}, e_3^1, e_3^2\}\rangle$ by issuing the event e_3^2 (see Fig. 7). To simulate this step, we need to show that it is possible



The traversal configuration TC_{b}

The event structure S_{b} and the selected execution X_{b}

Figure 7 The traversal configuration TC_{b}, the related event structure S_{b}, and the selected execution X_{b}.

to execute instructions of thread 2 and extend the event structure with a set of events Br_{b} matching these instructions. As we have already seen, the labels of the new events can differ from their counterparts in G—they only have to agree for the covered and issued events. In this case, we set $Br_{\mathsf{b}} = \{e_{11}^2, e_{21}^2, e_{31}^2\}$, and adding them to the event structure S_{a} gives us event structure S_{b} shown in Fig. 7.

In more detail, we need to build a run of thread-local semantics $prog(2) \xrightarrow{e_{11}^2} \xrightarrow{e_{21}^2} \xrightarrow{e_{31}^2} \sigma'$ such that (1) it contains events corresponding to all the events of thread 2 up to e_3^2 (*i.e.,* e_1^2, e_2^2, e_3^2) with the same location, type, and thread identifier and (2) any events corresponding to covered or issued events (*i.e.,* e_3^2) should also have the same value as the corresponding event in G.

Then, following the run of the thread-local semantics, we should extend the event structure S_{a} to S_{b} by adding new events Br_{b}, and ensure that the constructed event structure S_{b} is consistent (Def. 5) and simulates the configuration TC_{b}. In particular, it means that:

- for each read event in Br_{b} we need to pick a justification write event, which is either already present in S or po-preceed the read event;
- for each write event in Br_{b} we should determine its position in co order of the event structure.

Finally, we need to update the selected execution by replacing all events of thread 2 by the new events Br_{b}: $X_{\mathsf{b}} \triangleq X_{\mathsf{a}} \setminus S.\mathtt{thread}(\{2\}) \cup Br_{\mathsf{b}}$.

4.3.1 Justifying the New Read Events

In order to determine whence these read events should be justified (and hence what value they should return), we have adopted the approach of Podkopaev *et al.* [19] for a similar problem with certifying promises in the compilation proof from PS to IMM. The construction relies on several auxiliary definitions.

First, given an execution G and a traversal configuration $\langle C, I\rangle$, we define the set of *determined* events to be those events of G that must have equal counterparts in S. In particular, this means that S should assign to these events the same label as G, and thus the same reads-from source for the read events.

$$G.\mathtt{determined}_{\langle C,I\rangle} \triangleq C \cup I \cup dom((G.\mathtt{rf} \cap G.\mathrm{po})^? ; G.\mathrm{ppo} ; [I]) \cup codom([I] ; (G.\mathtt{rf} \cap G.\mathrm{po}))$$

Besides covered and issued events, the set of determined events also contains the ppo-prefixes of issued events, since issued events may depend on their values, as well as any internal reads reading from issued events, since their values are also determined by the issued events.

For the graph G and traversal configuration TC_{b}, the set of determined events contains events e_3^1, e_2^2, and e_3^2. (The events e_3^1 and e_3^2 are issued, whereas e_2^2 has a ppo edge to e_3^2.) In contrast, events e_1^1, e_2^1, and e_1^2 are not determined, since their corresponding events in S read/write a different value.

Second, we introduce the *viewfront* relation (vf) to contain all the writes that have been observed at a certain point in the graph. That is, the edge $\langle w, e\rangle \in G.\mathtt{vf}_{TC}$ indicates that the write w either happens before e, is read by a covered event happening before e, or is read by a determined read earlier in the same thread as e.

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

Figure 7 depicts three $G.\mathtt{vf}_{TC_{\mathsf{b}}}$ edges. Since $G.\mathtt{vf}_{TC}\,;G.\mathrm{po} \subseteq G.\mathtt{vf}_{TC}$, the other incoming viewfront edges to thread 2 can be derived. Note that there is no edge from e_2^1 to thread 2, since e_2^1 neither happens before any event in thread 2 nor is read by any determined read.

Finally, we construct the *stable justification* relation (sjf) that helps us justify the read events in Br_{b} in the event structure:

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

It relates a read event r to the co-last 'observed' write event with same location. Assuming that G is IMM-consistent, it can be shown that $G.\mathtt{sjf}$ agrees with $G.\mathtt{rf}$ on the set of determined reads.

$$G.\mathtt{sjf}_{TC};[G.\mathtt{determined}_{TC}] \subseteq G.\mathtt{rf}$$

For the graph G and traversal configuration TC_{b} shown in Fig. 7 the sjf relation coincides with the depicted vf edges: *i.e.,* we have $\langle \mathsf{Init}, e_1^1\rangle, \langle \mathsf{Init}, e_1^2\rangle, \langle e_3^1, e_2^2\rangle \in G.\mathtt{sjf}_{TC_{\mathsf{b}}}$.

Having $\mathtt{sjf}_{TC_{\mathsf{b}}}$ as a guide for values read by instructions in the certification run, we construct the steps of the thread-local operational semantics $prog(2) \rightarrow^* \sigma'$ using the receptiveness property of the thread's semantics, which essentially says that given an execution trace $\tau = e_1, \dots, e_n$ of the thread semantics, and a subset of events $K \subseteq \{e_1, \dots, e_{n-1}\}$ along that trace that have no ppo-successors in the graph, we arbitrarily change the values of read events in K, and there exist values for the write events in K such that the updated execution trace is also a trace of the thread semantics.[12]

The relation $\mathtt{sjf}_{TC_{\mathsf{b}}}$ is also used to pick justification writes for the read events in Br_{b}. We have proved that each sjf edge either starts in some issued event (of the previous traversal configuration) or it connects two events that are related by po:

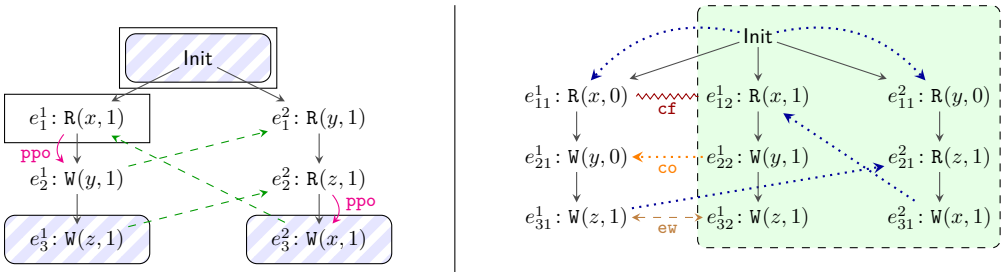
$$G.\mathtt{sjf}_{TC_{\mathsf{b}}} \subseteq [I_{\mathsf{a}}]\,;G.\mathtt{sjf}_{TC_{\mathsf{b}}} \cup G.\mathrm{po}$$

In the former case, thanks to the property 4 of our simulation relation, we can pick a write event from X_{a} corresponding to the issued write (*e.g.,* for Fig. 7, it is the event e_{31}^1, corresponding to the issued write e_3^1). In the latter case, we pick either the initial write or some $S_{\mathsf{b}}.\mathrm{po}$ preceding write belonging to Br_{b}.

4.3.2 Ordering the New Write Events

In order to pick the $S_b.\mathtt{co}$ position of the new write events in the updated event structure, we generally follow the original $G.\mathtt{co}$ order of the IMM graph. Because of the conflicting events,

[12] The formal definition of the receptiveness property is quite elaborate. For the detailed definition we refer the reader to the Coq development of IMM [7].



The traversal configuration TC_{c}

The event structure S_{c} and the selected execution X_{c}

Figure 8 The traversal configuration TC_{c}, the related event structure S_{c}, and the selected execution X_{c}.

however, it is not always possible to preserve the inclusion between the relations. This is why we relax the inclusion to $[\![S.\mathtt{co}]\!] \subseteq G.\mathtt{co}^?$ in property 12 of the simulation relation.

To see the problem let us return to the example. Suppose that the next traversal step covers the read e_1^1. To simulate this step, we build an event structure S_{c} (see Fig. 8). It contains the new events $Br_{\mathsf{c}} \triangleq \{e_{12}^1, e_{22}^1, e_{32}^1\}$.

Consider the write events e_{21}^1 and e_{22}^1 of the event structure. Since the events have different labels, we cannot make them $\mathtt{ew}$-equivalent. And since $S_{\mathsf{c}}.\mathtt{co}$ should be total among all writes to the same location (with respect to $S_{\mathsf{c}}.\mathtt{ew}$), we must put a $\mathtt{co}$ edge between these two events in one direction or another. Note that events e_{21}^1 and e_{22}^1 correspond to the same event e_2^1 in the graph, thus we cannot use the coherence order of the graph $G.\mathtt{co}$ to guide our decision.

In fact, the $\mathtt{co}$-order between these two events does not matter, so we could pick either direction. For the purposes of our proofs, however, we found it more convenient to always put the new events earlier in the $\mathtt{co}$ order (thus we have $\langle e_{22}^1, e_{21}^1\rangle \in S_{\mathsf{c}}.\mathtt{co}$). Thereby we can show that the $\mathtt{co}$ edges of the event structure ending in the new events, have corresponding edges in the graph: $[\![S_{\mathsf{c}}.\mathtt{co}\,;[Br_{\mathsf{c}}]]\!] \subseteq G.\mathtt{co}$.

Now consider the events e_{31}^1 and e_{32}^1. Since these events have the same label and correspond to the same event in G, we make them $\mathtt{ew}$-equivalent. In fact, this choice is necessary for the correctness of our construction. Otherwise, the new events Br_{c} would be deemed invisible, because of the $S_{\mathsf{c}}.\mathtt{cf} \cap (S_{\mathsf{c}}.\mathtt{jfe}\,;(S_{\mathsf{c}}.\mathtt{po} \cup S_{\mathsf{c}}.\mathtt{jf})^*)$ path between e_{31}^1 and e_{12}^1. Recall that only the visible events can be used to extract an execution from the event structure (Def. 7).

In general, assuming that $\mathcal{I}(prog, G, \langle C, I\rangle, S, X)$ holds, we attach the new write event e to an $S.\mathtt{ew}$ equivalence class represented by the write event w, *s.t.* (i) w has the same $\mathtt{s2g}$ image as e, *i.e.*, $\mathtt{s2g}(w) = \mathtt{s2g}(e)$; (ii) w belongs to X and its $\mathtt{s2g}$ image is issued, that is $w \in X \cap \lfloor\!\lfloor I \rfloor\!\rfloor$. If there is no such an event w, we put e $S.\mathtt{co}$-after events such that their $\mathtt{s2g}$ images are ordered $G.\mathtt{co}$-before $\mathtt{s2g}(e)$, and $S.\mathtt{co}$-before events such that their $\mathtt{s2g}$ images are equal to $\mathtt{s2g}(e)$ or ordered $G.\mathtt{co}$-after it. Note that thanks to property 9 of the simulation relation, that is $dom(S.\mathtt{jfe}) \subseteq dom(S.\mathtt{ew}\,;[X \cap \lfloor\!\lfloor I \rfloor\!\rfloor])$, our choice of $\mathtt{ew}$ guarantees that all new events will be visible.

4.3.3 Construction Overview

To sum up, to prove Lemma 3, we consider the events of $G.\mathtt{thread}(\{t\})$ where t is the thread of the event issued or covered by the traversal step $TC \longrightarrow TC'$, together with the

sjf relation determining the values of the read events. At this point, we can show that $\mathcal{I}$-conditions for the new configuration TC' hold for all events except for those in thread t.

Because of receptiveness, there exists a sequence of the thread steps $prog(t) \rightarrow^* \sigma'$ for some thread state σ' such that the labels on this sequence match the events $G.\texttt{thread}(\{t\})$ with the labels determined by sjf, and include an event with the same label as the one issued or covered by the traversal step $TC \longrightarrow TC'$.

We then do an induction on this sequence of steps, and add each event to the event structure S and to its selected subset of events X (unless already there), showing along the way that the $\mathcal{I}$-conditions also hold for the updated event structure, selected subset, and the events added. At the end, when we have considered all the events generated by the step sequence, we will have generated the event structure S' and execution X' such that $\mathcal{I}(prog, G, TC', S', X')$ holds.

5 Handling SC Accesses

In this section, we briefly describe the changes needed in order to handle the compilation of Weakestmo's sequentially consistent (SC) accesses. The purpose of SC accesses is to guarantee sequential consistency for the simple programming pattern that uses exclusively SC accesses to communicate between threads. As Lahav *et al.* [14] showed, however, their semantics is quite complicated because they can be freely mixed with non-SC accesses.

We first define an extension of IMM, which we call $\mathsf{IMM_{SC}}$. Its consistency extends that of IMM with an additional acyclicity requirement concerning SC accesses, which is taken directly from RC11-consistency [14, Definition 1].

▶ **Definition 10.** *An execution graph G is* $\mathsf{IMM_{SC}}$-consistent *if it is* IMM-*consistent [19, Definition 3.11] and $G.\mathtt{psc}_{\mathrm{base}} \cup G.\mathtt{psc}_{\mathrm{F}}$ is acyclic, where:*[13]

$$G.\mathtt{scb} \triangleq G.\mathrm{po} \cup G.\mathrm{po}|_{\neq G.\mathtt{loc}} ; G.\mathtt{hb} ; G.\mathrm{po}|_{\neq G.\mathtt{loc}} \cup G.\mathtt{hb}|_{=\mathtt{loc}} \cup G.\mathtt{co} \cup G.\mathtt{fr}$$

$$G.\mathtt{psc}_{\mathrm{base}} \triangleq ([G.\mathrm{E}^{\mathtt{sc}}] \cup [G.\mathrm{F}^{\mathtt{sc}}] ; G.\mathtt{hb}^{?}) ; G.\mathtt{scb} ; ([G.\mathrm{E}^{\mathtt{sc}}] \cup G.\mathtt{hb}^{?} ; [G.\mathrm{F}^{\mathtt{sc}}])$$

$$G.\mathtt{psc}_{\mathrm{F}} \triangleq [G.\mathrm{F}^{\mathtt{sc}}]; (G.\mathtt{hb} \cup G.\mathtt{hb}; G.\mathtt{eco}; G.\mathtt{hb}); [G.\mathrm{F}^{\mathtt{sc}}]$$

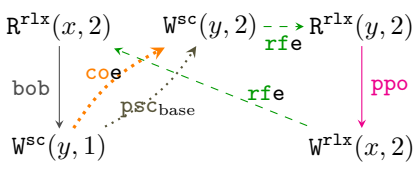
The scb, $\mathtt{psc}_{\mathrm{base}}$ and $\mathtt{psc}_{\mathrm{F}}$ relations were carefully designed by Lahav *et al.* [14] (and recently adopted by the C++ standard), so that they provide strong enough guarantees for programmers while being weak enough to support the intended compilation of SC accesses to commodity hardware. In particular, a previous (simpler) proposal in [2], which essentially includes $G.\mathtt{hb}$ between SC accesses in the relation required to be acyclic, is too strong for efficient compilation to the POWER architecture. Indeed, the compilation schemes to POWER do not enforce a strong barrier on hb-paths between SC accesses, but rather on $G.\mathrm{po} ; G.\mathtt{hb} ; G.\mathrm{po}$-paths between SC accesses.

▶ Remark 11. The full IMM model (*i.e.,* including release/acquire accesses and SC fences, as defined by Podkopaev *et al.* [19]) forbids cycles in $\mathtt{rfe} \cup \mathtt{ppo} \cup \mathtt{bob} \cup \mathtt{psc}_{\mathrm{F}}$, where bob is (similar to ppo) a subset of the program order that must preserved due to the presence of a memory fence or release/acquire access. Since $\mathtt{psc}_{\mathrm{F}}$ is already included in IMM's acyclicity constraint, one may consider the natural option of including $\mathtt{psc}_{\mathrm{base}}$ in that acyclicity constraint as well.

[13] In $\mathsf{IMM_{SC}}$, event labels include an "access mode", where sc denotes an SC access. The sets $G.\mathrm{E}^{\mathtt{sc}}$ consists of all SC accesses (reads, writes and fences) in G, and $G.\mathrm{F}^{\mathtt{sc}}$ consists of all SC fences in G.

However, it leads to a model that is too strong, as it forbids the following behavior:

$$\begin{array}{l} a := [x]^{\texttt{rlx}} \quad /\!/ 2 \\ [y]^{\texttt{sc}} := 1 \end{array} \;\Big\|\; [y]^{\texttt{sc}} := 2 \;\Big\|\; \begin{array}{l} b := [y]^{\texttt{rlx}} \quad /\!/ 2 \\ [x]^{\texttt{rlx}} := b \end{array}$$



This behavior is allowed by POWER (using any of the two intended compilation schemes for SC accesses; see Section 5.1.2).

Adapting the compilation from Weakestmo to $\mathsf{IMM_{SC}}$ to cover SC accesses is straightforward because the full definition of Weakestmo [6] does not have any additional constraints about SC accesses at the level of event structures. It only has an SC constraint at the level of extracted executions which is actually the same as in RC11, which we took as is for $\mathsf{IMM_{SC}}$.

5.1 Compiling $\mathsf{IMM_{SC}}$ to Hardware

In this section, we establish describe the extension of the results of [19] to support SC accesses with their intended compilation schemes to the different architectures.

As was done in [19], since $\mathsf{IMM_{SC}}$ and the models of hardware we consider are all defined in the same declarative framework (using execution graphs), we formulate our results on the level of execution graphs. Thus, we actually consider the mapping of $\mathsf{IMM_{SC}}$ execution graphs to target architecture execution graphs that is induced by compilation of $\mathsf{IMM_{SC}}$ programs to machine programs. Hence, roughly speaking, for each architecture $\alpha \in \{\mathsf{TSO}, \mathsf{POWER}, \mathsf{ARMv7}, \mathsf{ARMv8}\}$, our (mechanized) result takes the following form:

> If the α-execution-graph G_α corresponds to the $\mathsf{IMM_{SC}}$-execution-graph G, then α-consistency of G_α implies $\mathsf{IMM_{SC}}$-consistency of G.

Since the mapping from Weakestmo to $\mathsf{IMM_{SC}}$ (on the program level) is the *identity mapping* (Theorem 1), we obtain as a corollary the correctness of the compilation from Weakestmo to each architecture α that we consider. The exact notions of correspondence between G_α and G are presented in the technical appendix.

The mapping of $\mathsf{IMM_{SC}}$ to each architecture follows the intended compilation scheme of C/C++11 [16, 14], and extends the corresponding mappings of IMM from Podkopaev *et al.* [19] with the mapping of SC reads and writes. Next, we schematically present these extensions.

5.1.1 TSO

There are two alternative sound mappings of SC accesses to x86-TSO:

Fence after SC writes	Fence before SC reads
$(\!\lvert \mathtt{R}^{\mathtt{sc}} \rvert\!) \triangleq$ `mov`	$(\!\lvert \mathtt{R}^{\mathtt{sc}} \rvert\!) \triangleq$ `mfence;mov`
$(\!\lvert \mathtt{W}^{\mathtt{sc}} \rvert\!) \triangleq$ `mov;mfence`	$(\!\lvert \mathtt{W}^{\mathtt{sc}} \rvert\!) \triangleq$ `mov`
$(\!\lvert \mathtt{RMW}^{\mathtt{sc}} \rvert\!) \triangleq$ `(lock) xchg`	$(\!\lvert \mathtt{RMW}^{\mathtt{sc}} \rvert\!) \triangleq$ `(lock) xchg`

The first, which is implemented in mainstream compilers, inserts an `mfence` after every SC write; whereas the second inserts an `mfence` before every SC read. Importantly, one should *globally* apply one of the two mappings to ensure the existence of an `mfence` between every SC write and following SC read.

5.1.2 POWER

There are two alternative sound mappings of SC accesses to POWER:

Leading `sync`		Trailing `sync`	
$(\!\lvert \mathtt{R}^{\mathtt{sc}} \rvert\!)$	$\triangleq \mathtt{sync};(\!\lvert \mathtt{R}^{\mathtt{acq}} \rvert\!)$	$(\!\lvert \mathtt{R}^{\mathtt{sc}} \rvert\!)$	$\triangleq \mathtt{ld};\mathtt{sync}$
$(\!\lvert \mathtt{W}^{\mathtt{sc}} \rvert\!)$	$\triangleq \mathtt{sync};\mathtt{st}$	$(\!\lvert \mathtt{W}^{\mathtt{sc}} \rvert\!)$	$\triangleq (\!\lvert \mathtt{W}^{\mathtt{rel}} \rvert\!);\mathtt{sync}$
$(\!\lvert \mathtt{RMW}^{\mathtt{sc}} \rvert\!)$	$\triangleq \mathtt{sync};(\!\lvert \mathtt{RMW}^{\mathtt{acq}} \rvert\!)$	$(\!\lvert \mathtt{RMW}^{\mathtt{sc}} \rvert\!)$	$\triangleq (\!\lvert \mathtt{RMW}^{\mathtt{rel}} \rvert\!);\mathtt{sync}$

The first scheme inserts a `sync` before every SC access, while the second inserts an `sync` after every SC access. Importantly, one should *globally* apply one of the two mappings to ensure the existence of a `sync` between every two SC accesses.

Observing that `sync` is the result of mapping an SC-fence to POWER, we can reuse the existing proof for the mapping of IMM to POWER. To handle the leading `sync` (respectively, trailing `sync`) scheme we introduce a preceding step, in which we prove that splitting in the whole execution graph each SC access to a pair of an SC fence followed (preceded) by a release/acquire access is a sound transformation under $\mathsf{IMM}_{\mathsf{SC}}$. That is, this global execution graph transformation cannot make an inconsistent execution consistent:

▶ **Theorem 12.** *Let G be an execution graph such that*

$$[\mathtt{R}^{\mathtt{sc}} \cup \mathtt{W}^{\mathtt{sc}}] ; (G.\mathrm{po}' \cup G.\mathrm{po}' ; G.\mathtt{hb} ; G.\mathrm{po}') ; [\mathtt{R}^{\mathtt{sc}} \cup \mathtt{W}^{\mathtt{sc}}] \subseteq G.\mathtt{hb} ; [\mathtt{F}^{\mathtt{sc}}] ; G.\mathtt{hb},$$

where $G.\mathrm{po}' \triangleq G.\mathrm{po} \setminus G.\mathtt{rmw}$. Let G' be the execution graph obtained from G by weakening the access modes of SC write and read events to release and acquire modes respectively. Then, $\mathsf{IMM}_{\mathsf{SC}}$-consistency of G follows from IMM-consistency of G'.

Having this theorem, we can think about mapping of $\mathsf{IMM}_{\mathsf{SC}}$ to POWER as if it consists of three steps. We establish the correctness of each of them separately.

1. At the $\mathsf{IMM}_{\mathsf{SC}}$ level, we globally split each SC-access to an SC-fence and release/acquire access. Correctness of this step follows by Theorem 12.
2. We map IMM to POWER, whose correctness follows by the existing results of [19], since we do not have SC accesses at this stage.
3. We remove any redundant fences introduced by the previous step. Indeed, following the leading `sync` scheme, we will obtain `sync;lwsync;st` for an SC write. The `lwsync` is redundant here since `sync` provides stronger guarantees than `lwsync` and can be removed. Similarly, following the trailing `sync` scheme, we will obtain `ld;cmp;bc;isync;sync` for an SC read. Again, the `sync` makes other synchronization instructions redundant.

5.1.3 ARMv7

The ARMv7 model [1] is very similar to the POWER model with the main difference being that it has a weaker preserved program order than POWER. However, Podkopaev *et al.* [19] proved IMM to POWER compilation correctness without relying on POWER's preserved program order explicitly, but assuming the weaker version of ARMv7's order. Thus, their proof also establishes correctness of compilation from IMM to ARMv7.

Extending the proof to cover SC accesses follows the same scheme discussed for POWER, since two intended mappings of SC accesses for ARMv7 are the same except for replacing POWER's `sync` fence with ARMv7's `dmb`:

Leading `dmb`		Trailing `dmb`	
$(\!\lvert \mathtt{R}^{\mathtt{sc}} \rvert\!)$	$\triangleq \mathtt{dmb};(\!\lvert \mathtt{R}^{\mathtt{acq}} \rvert\!)$	$(\!\lvert \mathtt{R}^{\mathtt{sc}} \rvert\!)$	$\triangleq \mathtt{ldr};\mathtt{dmb}$
$(\!\lvert \mathtt{W}^{\mathtt{sc}} \rvert\!)$	$\triangleq \mathtt{dmb};\mathtt{str}$	$(\!\lvert \mathtt{W}^{\mathtt{sc}} \rvert\!)$	$\triangleq (\!\lvert \mathtt{W}^{\mathtt{rel}} \rvert\!);\mathtt{dmb}$
$(\!\lvert \mathtt{RMW}^{\mathtt{sc}} \rvert\!)$	$\triangleq \mathtt{dmb};(\!\lvert \mathtt{RMW}^{\mathtt{acq}} \rvert\!)$	$(\!\lvert \mathtt{RMW}^{\mathtt{sc}} \rvert\!)$	$\triangleq (\!\lvert \mathtt{RMW}^{\mathtt{rel}} \rvert\!);\mathtt{dmb}$

5.1.4 ARMv8

Since ARMv8 has added dedicated instructions to support C/C++-style SC accesses, we have established the correctness of a mapping employing these new instructions:

$$
\begin{aligned}
(\!|\mathtt{R}^{\mathtt{sc}}|\!) &\triangleq \mathtt{LDAR} \\
(\!|\mathtt{W}^{\mathtt{sc}}|\!) &\triangleq \mathtt{STLR} \\
(\!|\mathtt{FADD}^{\mathtt{sc}}|\!) &\triangleq \mathtt{L{:}LDAXR;STLXR;BC\ L} \\
(\!|\mathtt{CAS}^{\mathtt{sc}}|\!) &\triangleq \mathtt{L{:}LDAXR;CMP;BC\ Le;STLXR;BC\ L;Le{:}}
\end{aligned}
$$

We note that in this mapping, we follow Podkopaev *et al.* [19] and compile RMW operations to loops with load-linked and store-conditional instructions (`LDX`/`STX`). An alternative mapping for RMWs would be to use single hardware instructions, such as `LDADD` and `CAS`, that directly implement the required functionality. Unfortunately, however, due to a limitation of the current IMM setup and unclarity about the exact semantics of the `CAS` instruction, we are not able to prove the correctness of the alternative mapping employing these instructions. The problem is that IMM assumes that every po-edge from a RMW instruction is preserved, which holds for the mapping of `CAS` using the aforementioned loop, but not necessarily using the single instruction.

6 Related Work

While there are several memory model definitions both for hardware architectures [1, 10, 17, 21, 22] and programming languages [3, 4, 11, 15, 18, 20] in the literature, there are relatively few compilation correctness results [6, 9, 12, 14, 19, 25].

Most of these compilation results do not tackle any of the problems caused by po∪rf cycles, which are the main cause of complexity in establishing correctness of compilation mappings to hardware architectures. A number of papers (*e.g.,* [6, 12, 25]) consider only hardware models that forbid such cycles, such as x86-TSO [17] and "strong POWER" [13], while others (*e.g.,* [9]) consider compilation schemes that introduce fences and/or dependencies so as to prevent po∪rf cycles. The only compilation results where there is some non-trivial interplay of dependencies are by Lahav *et al.* [14] and by Podkopaev *et al.* [19].

The former paper [14] defines the RC11 model (repaired C11), and establishes a number of results about it, most of which are not related to compilation. The only relevant result is its pencil-and-paper correctness proof of a compilation scheme from RC11 to POWER that adds a fence between relaxed reads and subsequent relaxed writes, but not between non-atomic accesses. As such, the only po ∪ rf cycles possible under the compilation scheme involve a racy non-atomic access. Since non-atomic races have undefined semantics in RC11, whenever there is such a cycle, the proof appeals to receptiveness to construct a different acyclic execution exhibiting the race.

The latter paper [19] introduced IMM and used it to establish correctness of compilation from the "promising semantics" (PS) [12] to the usual hardware models. As already mentioned, IMM's definition catered precisely for the needs of the PS compilation proof, and so did not include important features such as sequentially consistent (SC) accesses. Our compilation proof shares some infrastructure with that proof—namely, the definition of IMM and traversals—but also has substantial differences because PS is quite different from Weakestmo. The main challenges in the PS proof were (1) to encode the various orders of the IMM execution graphs with the timestamps of the PS machine, and (2) to construct the certification runs for each outstanding promise. In contrast, the main technical challenge in the Weakestmo compilation proof is that event structures represent several possible executions

of the program together, and that Weakestmo consistency includes constraints that correlate these executions, allowing one execution to affect the consistency of another.

7 Conclusion

In this paper, we presented the first correctness proof of mapping from the Weakestmo memory model to a number of hardware architectures. As a way to show correctness of Weakestmo compilation to hardware, we employed IMM [19], which we extended with SC accesses, from which compilation to hardware follows.

Although relying on IMM modularizes the compilation proof and makes it easy to extend to multiple architectures, it does have one limitation. As was discussed in Section 5.1.4, IMM enforces ordering between RMW events and subsequent memory accesses, while one desirable alternative compilation mapping of RMWs to ARMv8 does not enforce this ordering, which means that we cannot prove soundness of that mapping via the current definition of IMM. We are investigating whether one can weaken the corresponding IMM constraint, so that we can establish correctness of the alternative ARMv8 mapping as well.

Another way to establish correctness of this alternative mapping to ARMv8 may be to use the recently developed Promising-ARM model [22]. Indeed, since Promising-ARM is closely related to PS [12], it should be relatively easy to prove the correctness of compilation from PS to Promising-ARM. Establishing compilation correctness of Weakestmo to Promising-ARM, however, would remain unresolved because Weakestmo and PS are incomparable [6]. Moreover, a direct compilation proof would probably also be quite difficult because of the rather different styles in which these models are defined.

References

1. Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014. URL: http://doi.acm.org/10.1145/2627752, doi:10.1145/2627752.
2. Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL 2016*, pages 634–648. ACM, 2016.
3. Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL 2011*, pages 55–66, New York, 2011. ACM. doi:10.1145/1925844.1926394.
4. John Bender and Jens Palsberg. A formalization of java's concurrent access modes. *Proc. ACM Program. Lang.*, 3(OOPSLA):142:1–142:28, October 2019. URL: http://doi.acm.org/10.1145/3360568, doi:10.1145/3360568.
5. Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *MSPC 2014*, pages 7:1–7:6. ACM, 2014. doi:10.1145/2618128.2618134.
6. Soham Chakraborty and Viktor Vafeiadis. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.*, 3(POPL):70:1–70:27, 2019. doi:10.1145/3290383.
7. The Coq development of IMM, available at http://github.com/weakmemory/imm, 2019.
8. Will Deacon. The ARMv8 application level memory model, 2017. URL: https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat.
9. Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. Bounding data races in space and time. In *PLDI 2018*, pages 242–255, New York, 2018. ACM. URL: http://doi.acm.org/10.1145/3192366.3192421, doi:10.1145/3192366.3192421.
10. Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *POPL 2016*, pages 608–621, New York, 2016. ACM. URL: http://doi.acm.org/10.1145/2837614.2837615, doi:10.1145/2837614.2837615.

11 Alan Jeffrey and James Riely. On thin air reads towards an event structures model of relaxed memory. In *LICS 2016*, pages 759–767, New York, 2016. ACM. URL: http://doi.acm.org/10.1145/2933575.2934536, doi:10.1145/2933575.2934536.

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

13 Ori Lahav and Viktor Vafeiadis. Explaining relaxed memory models with program transformations. In *FM 2016*. Springer, 2016. doi:10.1007/978-3-319-48989-6_29.

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

15 Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL 2005*, pages 378–391, New York, 2005. ACM. doi:10.1145/1040305.1040336.

16 C/C++11 mappings to processors, 2016. URL: http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.

17 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 391–407, Heidelberg, 2009. Springer.

18 Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *POPL 2016*, pages 622–633, New York, 2016. ACM. doi:10.1145/2837614.2837616.

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

20 Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. Operational aspects of C/C++ concurrency. *CoRR*, abs/1606.01400, 2016. URL: http://arxiv.org/abs/1606.01400.

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

22 Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1–15, New York, NY, USA, 2019. ACM. URL: http://doi.acm.org/10.1145/3314221.3314624, doi:10.1145/3314221.3314624.

23 The RISC-V instruction set manual. volume i: Unprivileged ISA, 2018. Available at https://github.com/riscv/riscv-isa-manual/releases/download/draft-20180731-e264b74/riscv-spec.pdf [Online; accessed 23-August-2018].

24 RISC-V: herd vs. operational models, 2018. URL: http://diy.inria.fr/cats7/riscv/.

25 Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013. doi:10.1145/2487241.2487248.