Reconciling Event Structures with Modern Multiprocessors

Evgenii Moiseenko¹,³, Anton Podkopaev²,³,⁶, Ori Lahav⁴, Orestis Melkonian⁵, and Viktor Vafeiadis⁶

¹ St. Petersburg University, Russia
² National Research University Higher School of Economics, Russia
³ JetBrains Research, Russia
⁴ Tel Aviv University, Israel
⁵ University of Edinburgh, UK
⁶ MPI-SWS, Germany

Abstract. Weakestmo is a recently proposed memory consistency model that uses event structures to resolve the infamous “out-of-thin-air” problem. Although it has been shown to have important benefits over other memory models, its established compilation schemes are suboptimal in that they add more fences than necessary.

In this paper, we prove the correctness in Coq of the intended compilation schemes for Weakestmo to a range of hardware memory models (x86, POWER, ARMv7, ARMv8, RISC-V). Our proof is the first that establishes correctness of compilation of an event-structure-based model that forbids “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.

Keywords: Weak Memory Consistency IMM Weakestmo.

1 Introduction

A large body of research on weak memory models has recently been devoted to developing models that allow 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 [16,10,4,12]. For example, under the assumption that locations $x$ and $y$ are initialized by 0, it is desirable to allow annotated outcome $a = b = 1$ for LB-fake, while forbidding it for LB-data.

\[
\begin{align*}
  a &:= [x] \quad \text{(LB-fake)} \\
  [y] &:= 1 + a * 0 \\
  [x] &:= b \\
  b &:= [y] \\
\end{align*}
\]

The most established model which meets the desideratum is the promising semantics of Kang et al. [11] (henceforth, PS), an operational model based on
timestamps and ‘promises’ of future writes. Podkopaev et al. [17] recently proved the correctness of compilation from PS to hardware memory models by introducing an intermediate memory model, called IMM [17], that abstracts over the major existing hardware models (x86-TSO [15], POWER [1], ARMv7 [1], ARMv8 [19], RISC-V [21,22]), thereby modularizing the proof.

Nevertheless, PS has some drawbacks pertaining to its compilation: (1) it does not support sequential consistent (SC) accesses, the default C/C++ atomic access mode; (2) its compilation of read-modify-write (RMW) operations (e.g., compare-and-swap and fetch-and-add) to ARMv8 requires an extra fence. Moreover, PS is not very flexible in that it is not easy to adapt its definition.

For these reasons, Chakraborty and Vafeiadis [5] introduced the Weakestmo memory model based on event structures. Being largely declarative, Weakestmo is flexible and supports SC accesses, and was conjectured to support optimal compilation to hardware memory models, i.e., not requiring any fences or fake dependencies for relaxed accesses (incl. RMWs). Because of the difficulty of establishing such a result, Chakraborty and Vafeiadis only established correctness of the intended compilation scheme to x86-TSO, as well as correctness of two suboptimal compilation schemes to POWER and ARMv7 that involve more expensive fences than intended. Compilation to ARMv8 is not established.

In this paper, we formalize Weakestmo in Coq and establish the correctness of the intended compilation schemes from Weakestmo to the aforementioned hardware architectures with a mechanized Coq proof of about 30K lines on top of an existing infrastructure of about 19K lines. Our proof also revealed some minor deficiencies in the Weakestmo model, which we have addressed.

To the best of our knowledge, our proof is the first mechanized compilation proof of an event-structure-based memory model, as well as the first mechanized compilation proof of a weak memory model supporting SC accesses. The latter, perhaps counter-intuitively, are not easy to support and have had a history of wrong compilation correctness arguments (see [12] for details). To achieve this, we introduce IMM_{SC}, an extension of IMM with SC accesses, which we consider as a valuable secondary contribution, since IMM_{SC} is already used for proving the correctness of compilation of the OCaml and JavaScript models. The compilation proof is structured as shown in Fig. 1 with the bold arrow representing the main result, the others being extensions of previous results, and double arrows denoting results for two compilation schemes.

Outline We start with an overview of IMM, Weakestmo, and our compilation proof (§2); we then present Weakestmo (§3), IMM_{SC} (§4), a simulation relation used in compilation correctness proof (§5), and the proof of the simulation step
Reconciling Event Structures with Modern Multiprocessors

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

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

(§6). The associated proof scripts can be found at http://github.com/weakmemory/weakestmoToImm and http://github.com/weakmemory/imm.

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 I:

\[
\begin{align*}
  a &:= [x] / / 1 \\
  [y] &:= 1 \\
  b &:= [y] / / 1 \\
  [x] &:= b
\end{align*}
\]  

(LB)

2.1 An Informal Introduction to IMM and IMM$_{SC}$

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 $G_{LB}$, an IMM execution graph of LB corresponding to an execution yielding the annotated behavior.

Vertices of execution graphs, called events, represent memory accesses which are either initialization of memory or generated by execution of program instructions. Each non-initialization 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 $W(x, 0)$ for each location $x$ used in the program; for conciseness, however, we depict it as a single event Init.

The edges 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. The data relation represents a syntactic data dependency between events of the same thread (e.g., a write storing the value read by a prior read). In examples, we depict only immediate po edges and omit marking po edges between events connected by another thread-local relation like data.

As a declarative memory model, IMM employs a consistency predicate to define which executions are allowed (i.e., are IMM-consistent). The predicate
is defined as a collection of constraints forbidding cycles of different shapes in executions graphs. For example, IMM forbids cycles consisting only of rf and data edges. That is, the execution in Fig. 2b, which represents the annotated behavior of LB-fake and LB-data, is not IMM-consistent. In contrast, $G_{LB}$ is IMM-consistent, and so IMM allows the annotated behavior of LB, but forbids those of LB-fake and LB-data.

Execution graphs of a program are constructed in three steps in IMM. First, sequential executions are built for each thread in accordance to a thread-local semantics, which non-deterministically picks for each read access the value being read (among the set of all possible values). Second, the executions of different threads are combined to a single complete execution graph. An execution graph is complete if each read event in it is connected to some write of the same location and value by the rf relation. Third, the execution graphs are filtered by the IMM-consistency predicate. The IMM-consistent ones form the program’s semantics.

Since Weakestmo supports C11-style sequential consistent (SC) accesses, which IMM does not support, we extend IMM with SC accesses following their axiomatization by Lahav et al. [12]. Our extended model, IMM$_{SC}$, adds some constraints in its consistency definition for SC accesses. The extension is conservative in that programs without SC accesses have the same semantics under IMM and IMM$_{SC}$.

2.2 An Informal Introduction to Weakestmo

We move on to the Weakestmo model, 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 builds for a program. The event structures themselves are built operationally. For example, a sequence of event structures which Weakestmo constructs for LB’s thread I is presented in Fig. 3.

The initial event structure consists of only initial events. Then, Weakestmo may add an event representing execution of the first instruction of a program’s thread. Then, Weakestmo may execute the second instruction of this thread or the first instruction of another thread, etc. Fig. 3a depicts the event structure $S_a$ obtained from the initial event structure by executing $a := [x]$ in LB’s thread I. As a result of the instruction execution, a read event $e_{11}^R: R(x, 0)$ is added.

Whenever the event added is a read, Weakestmo select a write event to the same location to justify the value returned by the read. 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 $e_{11}^R$ 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. As a consequence of how Weakestmo event structure are constructed, $po \cup jf$ is guaranteed to be acyclic.

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

---

7 For a detailed formal description of the graphs and their construction process we refer the reader to [17, §2.2].
track syntactic dependencies, e.g., $S_d$ in Fig. 3d does not contain a data edge between $e_{12}^1$ and $e_{12}^2$. The reason is that Weakestmo is a programming-language-level memory model and supports optimizations removing fake dependencies.

The next step (Fig. 3c) 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 I and adds a new event $e_{12}^1$ justified by a different write event. We say that this new event conflicts with $e_{11}^1$ because they cannot both occur in a single execution.

Technically, conflicting events are represented by a symmetric “conflict” relation $\text{cf}$. 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 if they are not ordered by po. By construction, $\text{cf}$ “extends downwards”: po-successors of conflicting events are also in conflict with one another.

The final construction step (Fig. 3f) demonstrates another feature of Weakestmo event structures. Conflicting write events writing the same value to the same lo-
cation may be announced equal writes, i.e., connected by an equivalence relation \( \text{ew} \), e.g., events \( e_1^{22} \) and \( e_1^{12} \) in \( S_f \).

The \( \text{ew} \) relation is used to define Weakestmo’s version of the reads-from relation, \( \text{rf} \), which relates a read to all (non-conflicted) writes equal to the write justifying the read. For example, \( e_2^{11} \) reads from both \( e_2^{21} \) and \( e_2^{12} \).

The Weakestmo’s \( \text{rf} \) relation is used for extraction of program executions. An execution graph \( G \) is extracted from an event structure \( S \) denoted \( S \triangleright G \) if, among other requirements, \( G \) is a maximal conflict-free subset of \( S \) such that each read event in \( G \) reads from some write in \( G \) according to \( S_{\text{rf}} \). There are two execution graphs which could be extracted from \( S_f \): \{Init, \( e_1^{11}, e_2^{12}, e_2^{11}, e_2^{21} \)\} and \{Init, \( e_1^{12}, e_2^{22}, e_1^{11}, e_2^{21} \)\} representing outcomes \( a = 0 \land b = 1 \) and \( a = b = 1 \) respectively.

### 2.3 Weakestmo to IMM\(_{SC}\) Compilation: High-Level Proof Structure

In this paper, we assume that Weakestmo is defined for the same assembly language as IMM (see [17, Fig. 2]) extended with SC accesses and refer to the language as \( L \). Having that, we show the correctness of the identity mapping as a compilation scheme from Weakestmo to IMM\(_{SC}\) in the following theorem.

**Theorem 1.** Let prog be a program in \( L \), and \( G \) be an IMM\(_{SC}\)-consistent execution graph of prog. Then there exists an event structure \( S \) of prog under Weakestmo such that \( S \triangleright G \).

---

8 In this paper, we take \( \text{ew} \) to be reflexive, whereas it is irreflexive in [5]. Our \( \text{ew} \) is the reflexive closure of [5]’s one.
To prove the theorem, we show that Weakestmo may construct the needed event structure in a step by step fashion by following a traversal of IMM$_SC$-consistent execution graph from [17, §§6.7].

A traversal of an IMM$_SC$-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, $(C, I)$, 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 $G_{LB}$ of LB from Fig. 2a, with the issued set marked by $\bigcirc$ and the covered set marked by $\Box$.

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. For example, a traversal cannot issue $e_2^2 \wedge (x, 1)$ before issuing $e_1^2 \wedge (y, 1)$ in Fig. 4, or cover $e_1^1 \wedge (x, 1)$ before issuing $e_2^1 \wedge (x, 1)$. These constraints allow to simulate the traversal by constructing a Weakestmo event structure.

According to [17, Prop. 6.5], every IMM$_SC$-consistent execution graph $G$ has a full traversal of the following form:

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

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

Our simulation proof is divided into the following three lemmas, which establish a simulation relation, $\mathcal{I}(\text{prog}, G, TC, S, X)$, between the current traversal configuration $TC$ of execution $G$ and the current event structure’s state $(S, X)$, where $X$ is a subset of events corresponding to a particular execution graph extracted from the event structure $S$.

**Lemma 1.** Let $\text{prog}$ be a program of $L$, and $G$ be an IMM$_SC$-consistent execution graph of $\text{prog}$. Then $\mathcal{I}(\text{prog}, G, TC_{\text{init}}(G), S_{\text{init}}(\text{prog}), S_{\text{init}}(\text{prog}).E)$ holds.

**Lemma 2.** If $\mathcal{I}(\text{prog}, G, TC, S, X)$ and $G \vdash TC \rightarrow TC'$ hold, then there exist $S'$ and $X'$ such that $\mathcal{I}(\text{prog}, G, TC', S', X')$ and $S \rightarrow^* S'$ hold.

**Lemma 3.** If $\mathcal{I}(\text{prog}, G, TC_{\text{final}}(G), S, X)$ 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 $G \vdash TC_{\text{init}}(G) \rightarrow^* TC_{\text{final}}(G)$. Lemma 1 serves as the base case, and Lemma 2 is the induction step simulating each traversal step with a number of event structure construction steps, and Lemma 3 concludes the proof.

---

9 For readers familiar with PS [11], issuing a write events corresponds to promising a message, and covering an event to normal execution of an instruction.
The proofs of Lemmas 1 and 3 are technical but fairly straightforward. In contrast, Lemma 2 is much more difficult to prove. As we will see, simulating a traversal step sometimes requires constructing a new branch in the event structure, i.e., adding multiple events. For this reason, we introduce an intermediate simulation relation that holds throughout that construction (see §6).

2.4 Weakestmo to IMMSC Compilation Correctness by Example

Before presenting any formal definitions, we conclude this overview section by showcasing the construction used in the proof of Lemma 2 on execution graph $G_{LB}$ 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 $X_f \triangleq \{\text{Init}, e_{12}, e_{22}, e_{11}, e_{21}\}$, which we call a simulated execution, marked by $\emptyset$ is a maximal conflict-free subset of $S_f$ and all read events in $X_f$ are in codomain of $S_f$. Then, by definition, $X_f$ is extracted from $S_f$. Also, an execution graph induced by $X_f$ is isomorphic to $G_{LB}$. That is, construction of $S_f$ for $LB$ shows that in Weakestmo it is possible to observe the same behavior as $G_{LB}$. Now, we explain how we construct $S_f$ and choose $X_f$.

During the simulation, we maintain the relation $I(prog, G, TC, S, X)$ 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, the relation states that all issued and covered events of $TC$ have exact counterparts in $X$. Also, we require $X$ to be extracted from $S$.

The initial event structure and $X_{\text{init}}$ consist of only initial events. Then, following issuing of event $e_{12} : W(y, 1)$ in $TC_{a}$ (see Fig. 4a), we need to add a branch to the event structure s.t. it has $W(y, 1)$ in it. Since Weakestmo requires to add events according to the program order, we first need to add a read event related to ‘$a := [x]$’ of LB’s thread I. 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 $e_{11}$ 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 $e_{21} : W(y, 1)$ could be added po-after $e_{11}$ (see Fig. 3b), and $I(LB, G_{LB}, TC_{a}, S_{b}, X_{b})$ holds for $X_{b} = \{\text{Init}, e_{11}, e_{21}\}$.

Next, we need to simulate the second traversal step (see Fig. 4b), which issues $W(x, 1)$. As with the previous step, we first need to add a read event related to the first read instruction of LB’s thread II (see Fig. 3c). However, unlike the previous step, the added event $e_{21}$ has to get value 1, since there is a dependency between instructions in thread II. As we mentioned earlier, the traversal strategy guarantees that $e_{2} : W(y, 1)$ is issued at the moment of issuing $e_{2} : W(x, 1)$, so there is the corresponding event in the event structure to justify
the read event \( e^2_{21} \) from. Now, the write event \( e^2_{21} : \hat{w}(y, 1) \) representing \( e^2_2 \) could be added to the event structure (see Fig. 3d) and \( I(LB, G_{LB}, TC_e, S_{e}, X_{e}) \) holds for \( X_{e} = \{ \text{init}, e^3_{11}, e^1_{21}, e^3_{11}, e^2_{21} \} \).

In the third traversal step (see Fig. 4c), the read event \( e^1_1 : \hat{r}(x, 1) \) is covered. To have a representative event for \( e^1_1 \) in the event structure, we add \( e^1_{12} \) (see Fig. 3c). It is justified from \( e^2_{21} \), which writes the needed value 1. Also, \( e^1_{12} \) represents an alternative to \( e^1_{11} \) execution of the first instruction of thread I, so the events are in conflict.

However, we cannot choose a simulated execution \( X \) related to \( TC_e \) and \( S_e \) by the simulation relation since \( X \) has to contain \( e^1_{12} \) and a representative for \( e^2_2 : \hat{w}(y, 1) \) (in \( S_e \) it is represented by \( e^3_{21} \)) 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 \( e^1_2 : \hat{w}(y, 1) \). Now, the simulated execution contains everything needed, \( X_{f} = \{ \text{init}, e^1_{12}, e^2_{22}, e^3_{11}, e^2_{21} \} \).

Since \( X_{f} \) has to be extracted from \( S_f \), every read event in \( X \) has to be connected via an \( rf \) edge to an event in \( X \).\(^{10}\) To preserve the requirement, we connect the newly added event \( e^1_{12} \) and \( e^2_{21} \) via an \( ew \) edge, i.e., marked them to be equal writes.\(^{11}\) This induces an \( rf \) edge between \( e^1_{22} \) and \( e^3_{11} \). That is, \( I(LB, G_{LB}, TC_e, S_{f}, X_{f}) \) holds.

To simulate the latter traversal steps (see Figures 4d to 4f), we don’t need to modify \( S_f \) since the execution graph associated with \( X_{f} \) is isomorphic to \( G_{LB} \).

That is, in the proof we just need to show that, first, \( I(LB, G_{LB}, TC_e, S_{f}, X_{f}) \) holds, then \( I(LB, G_{LB}, TC_e, S_{f}, X_{f}) \) holds, and, finally, \( I(LB, G_{LB}, TC_e, S_{f}, X_{f}) \) holds.

### 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 formally.

**Notation** Given relations \( R_1 \) and \( R_2 \), we write \( R_1 : R_2 \) for their sequential composition. Given relation \( R \) we write \( R^\alpha, R^\beta \) and \( R^\gamma \) to denote its reflexive, transitive and reflexive-transitive closures. For a set \( A \), we write \([A]\) to denote the identity relation on \( A \) (that is, \([A] = \{(a, a) \mid a \in A\}\)). 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 \( f \), we denote by \( \equiv_f \) the set of \( f \)-equivalent elements: \( \equiv_f = \{ (a, b) \mid f(a) = f(b) \} \). In addition, given a relation \( R \), we denote by \( R|_{=f} \) the restriction of \( R \) to \( f \)-equivalent elements \( (R|_{=f}) = f \cap \equiv_f \), and by \( R|_{\neq f} \) be the restriction of \( R \) to non-\( f \)-equivalent elements \( (R|_{\neq f}) = R \setminus \equiv_f \).

\(^{10}\) 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.

\(^{11}\) Note that we could have left \( e^1_{12} \) without any outgoing \( ew \) edges since the choice of equal writes for newly added events in Weakestmo is non-deterministic. However, that would not preserve the simulation relation.
3.1 Events, Threads and Labels

Events, \( e \in E \), and thread identifiers, \( t \in Tid \), are represented just by unique natural numbers. We treat the thread with identifier 0 as the initialization thread. We let \( x \in \text{Loc} \) to range over locations, and \( v \in \text{Val} \) over values.

Each memory access has a mode which is either relaxed (rlx), release (rel), acquire (acq), acquire-release (acqrel), or sequentially-consistent (sc).\(^\text{12}\) The modes are partially ordered by \( \sqsubseteq \) as follows:

\[
\text{rlx} \sqsubseteq \text{rel} \sqsubseteq \text{acq} \sqsubseteq \text{acqrel} \sqsubseteq \text{sc}
\]

A label, \( l \in \text{Lab} \), takes one of the following forms:

- \( R^o(x, v) \) — a read of value \( v \) from location \( x \) with mode \( o \in \{ \text{rlx, acq, sc} \} \).
- \( W^o(x, v) \) — a write of value \( v \) to location \( x \) with mode \( o \in \{ \text{rlx, rel, sc} \} \).
- \( F^o \) — a fence with mode \( o \in \{ \text{acq, rel, acqrel, sc} \} \).

Given a label \( l \) the functions typ, loc, val, mod return (when applicable) its type (i.e., R, W or F), location, value and mode correspondingly. By abuse of notation, we also use \( R, W, F \) for the set of all events with the corresponding type as well as, for example, \( RW \) for \( R \cup W \). We also use subscripts and superscripts to further restrict this set (e.g., \( W_\text{rel}^x \) denotes the set of write events operating on location \( x \) with mode at least as strong as \( \text{rel} \)).

3.2 Event Structures

An event structure \( S \) is a tuple \( \langle E, \text{tid}, \text{lab}, \text{po}, \text{rmw}, \text{jf}, \text{ew}, \text{co}, \text{K}_{\text{init}}, \text{K} \rangle \) where:

- \( E \) is a set of events.
- \( \text{tid} : E \to Tid \) is a function that assigns a thread identifier to every event. We treat events with the thread identifier equal to 0 as initialization events and denote them as \( \text{Init} \), that is \( \text{Init} \triangleq \{ e \in E \mid \text{tid}(e) = 0 \} \).
- \( \text{lab} : E \to \text{Lab} \) is a function that assigns a label to every event.
- \( \text{po} \subseteq E \times E \) is a strict partial order on events, called program order, that tracks their precedence in the control flow of the program. Initialization events are po-before all other events and po edges relate non-initialization events only when they are from the same thread. \( \text{po} \) does not necessarily totally order all events of the same thread. Non-initialization events of the same thread, that are not related by program order, are called conflicting events. The corresponding binary relation cf is defined as follows:

\[
\text{cf} \triangleq [E \setminus \text{Init}] ; ( =_{\text{tid}} \setminus (\text{po} \cup \text{po}^{-1}) ) ; [E \setminus \text{Init}]
\]

\(^{12}\) For the purposes of this paper, Weakest non-atomic accesses [5] are treated as relaxed accesses, since they are compiled in the same way.
We say that an event $e_1$ is an immediate predecessor of $e_2$ if $e_1$ is a predecessor of $e_2$ and there is no event between them.

$$\text{po}_{\text{in}} \triangleq \text{po} \setminus (\text{po} \circ \text{po})$$

We also define the notion of immediate conflict:\textsuperscript{13}

$$\text{cf}_{\text{in}} \triangleq \text{cf} \cap (\text{po}_{\text{in}}^{-1} ; \text{po}_{\text{in}})$$

- $\text{rmw} \subseteq [E \cap R] ; (\text{po}_{\text{in}} \cap \equiv_{\text{loc}} ; [E \cap W])$ is the read-modify-write pairing.
- $\text{jf} \subseteq [E \cap W] ; (\equiv_{\text{loc}} \cap \equiv_{\text{val}} ; [E \cap R])$ is the justified from relation, which relates a write event to the reads it justifies. We require that a read not be justified by a conflicting write (i.e., $\text{jf} \cap \text{cf} \subseteq \emptyset$) and $\text{jf}^{-1}$ be functional (i.e., whenever $\langle w_1, r \rangle, \langle w_2, r \rangle \in \text{jf}$, then $w_1 = w_2$). We also define the notions of internal and external justification,
  - $\text{ji} \triangleq \text{jf} \cap \text{po}$
  - $\text{je} \triangleq \text{jf} \setminus \text{po}$
- $\text{ew} \subseteq [E \cap W] ; (\equiv_{\text{loc}} \cap \equiv_{\text{val}} \cap ([\text{wr} ; W] ; [\text{wr} ; W]) ; [E \cap W])$ is an equivalence relation called the equal-writes relation. Note that equal writes have the same location and value, and non-reflexive edges of $\text{ew}$ relate only conflicting writes.
- $\text{co} \subseteq [E \cap W] ; (\equiv_{\text{loc}} \setminus \text{ew}) ; [E \cap W]$ 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., $\text{ew} ; \text{co} ; \text{ew} \subseteq \text{co}$) and total with respect to $\text{ew}$ on writes to the same location:

$$\forall x \in \text{Loc}. \forall w_1, w_2 \in \tilde{W_x}. \langle w_1, w_2 \rangle \in \text{ew} \cup \text{co} \cup \text{co}^{-1}$$

- $K_{\text{init}}$ and $K$ components are related to the process of the event structure construction, which is explained in more detail later in this section. They are functions whose codomain is a set of thread states.
  - $K_{\text{init}} : \text{Tid} \rightarrow \text{ThreadState}$ is a function that given a thread identifier $t$ returns the initial state $\sigma_0$ of the thread.
  - $K : E \setminus \text{dom}(\text{rmw}) \rightarrow \text{ThreadState}$ is a function that assigns state $\sigma$ to every event $e$, that is not a read part of some read-modify-write pair. This state corresponds to the thread state after the effect of the event $e$ has been performed.

Given an event structure $S$, we use $S.x$ notation to refer to its components (e.g., $S.E$, $S.po$ etc.). For a set $A$ of events, we write $S.A$ for the set $A \cap S.E$ (e.g., $S.W_x$). Further, for $e \in S.E$, we write $S.\text{typ}(e)$ to retrieve $\text{typ}(S.\text{lab}(e))$. Similar notation is used for the functions $\text{loc}$, $\text{val}$, and $\text{mod}$. Given a set of thread identifiers $T$, we also use notation $S.\text{thread}(T)$ to denote the set of events belonging to one of the threads from $T$, i.e., $S.\text{thread}(T) \triangleq \{ e \in S.E \mid S.\text{tid}(e) \in T \}$. By abuse of notation we often write $S.\text{thread}(t)$ instead of $S.\text{thread}(\{t\})$, assuming $t$ is a single thread identifier.

\textsuperscript{13} Our definition of immediate conflicts differs from that of [5] and is easier to work with. The two definitions are equivalent if the set of initialization events is non-empty.
3.3 Derived Sets and Relations

First, the reads-from relation, $S_{rf}$, of a Weakestmo event structure is derived. It is defined as an extension of $S_{jf}$ to all $S_{ew}$-equivalent writes. Note that unlike $S_{jf}^{-1}$ the $S_{rf}^{-1}$ relation is not functional.

$$S_{rf} \triangleq (S_{ew} ; S_{jf}) \setminus S_{cf}$$

The relation $S_{fr}$, called from-reads or reads-before, places read events before subsequent writes.

$$S_{fr} \triangleq S_{rf}^{-1} ; S_{co}$$

The extended coherence $S_{eco}$ is a strict partial order, that orders write-write, write-read and read-write pairs of events operating on the same location.

$$S_{eco} \triangleq (S_{co} \cup S_{rf} \cup S_{fr})^+$$

We observe that in our model, $S_{eco}$ is equal to $rf \cup co : rf \cup fr : rf$, similar to the corresponding definitions on execution graphs in the literature.\(^{14}\)

Next, we define the synchronizes-with $S_{sw}$ and happens-before $S_{hb}$ relations, using auxiliary notions of release sequence $S_{rs}$ and release prefix $S_{release}$. These definitions coincides with the conventional definitions except that in the Weakestmo case the $jf$ relation is used in place of $rf$.

$$S_{rs} \triangleq [S_W] ; (S_{po} \cup S_{s1co})^2 ; (S_{jf} ; S_{rmw})^*$$

$$S_{release} \triangleq [S_E^{rel}] ; ([S_F] ; S_{po})^2 ; S_{rs}$$

$$S_{sw} \triangleq S_{release} ; S_{jf} ; (S_{po} ; [S_F])^2 ; [S_E^{seq}]$$

$$S_{hb} \triangleq (S_{po} \cup S_{sw})^+$$

We say that two events are in extended conflict if they happen after some conflicting events.

$$S_{ecf} \triangleq (S_{hb}^{-1})^2 ; S_{cf} ; S_{hb}$$

The last ingredient that we need for event structure consistency is the notion of visible events. We define it in a few steps. First, consider an event and all the write events, that were used to externally justify it or one of its $S_{po} \cup S_{jf}$ ancestors. The relation $S_{jfe} ; (S_{po} \cup S_{jf})^*$ defines this connection formally. Next, consider only those writes that are in conflict with the event they ‘recursively’ justify: $S_{cf} \cap S_{jfe} ; (S_{po} \cup S_{jf})^*$. We say that event is visible if all such writes are $ew$-equivalent to some write event in the same control-flow branch of the program.\(^{15}\)

$$S_{Vis} \triangleq \{ e \in S_E \mid S_{cf} \cap (S_{jfe} ; (S_{po} \cup S_{jf})^*) ; [e] \subseteq S_{ew} ; (po \cup po^{-1})^2 \}$$

\(^{14}\) This equivalence equivalence does not hold in the original Weakestmo model [5]. To make the equivalence hold, we made $ev$ transitive, and required $ev ; co ; ev \subseteq co$.

\(^{15}\) Note, that in [5] 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.
3.4 Event Structure Consistency

Similarly to the axiomatic style of the memory model definitions, Weakestmo further restricts the semantics of the program by the requirement on the event structure to satisfy the consistency predicate.

**Definition 1.** An event structure \( S \) is said to be consistent if the following conditions hold.

- \( S \text{-ecf} \) is irreflexive. \( \text{(ecf-IRREFLEXIVITY)} \)
- \( S \text{-jf} \cap S \text{-ecf} \equiv \emptyset \) \( \text{(jf-NON-CONFLICT)} \)
- \( \text{dom}(S \text{-jfe}) \subseteq S \text{-Vis} \) \( \text{(jfe-VISIBLE)} \)
- \( S \text{-hb}; S \text{-eco}^{-1} \) is irreflexive \( \text{(COHERENCE)} \)
- \( \text{dom}(S \text{-cfimm}) \subseteq S \text{-R} \) \( \text{(cfimm-READ)} \)
- \( S \text{-jf}; S \text{-cfimm}; S \text{-jfe}^{-1}; S \text{-ew} \) is irreflexive \( \text{(cfimm-JUSTIFICATION)} \)

In brief, consistency requires that (1) no event happen after two conflicting events, (2) reads not be justified by a write in extended conflict, (3) reads be justified only by po-prior or visible events, (4) the execution be coherent, (5) immediate conflicts arise only because of read events, and (6) there exist no duplicate read events (i.e., in immediate conflict and justified by equal writes).

3.5 Execution Extraction

We move on to the extraction of executions from an event structure.

First, we define an execution graph.

**Definition 2.** An execution graph \( G \) is a tuple \( \langle E, \text{tid}, \text{lab}, \text{po}, \text{rmw}, \text{rf}, \text{co} \rangle \) where its components are defined similarly as in case of an event structure with the following exceptions:

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

All derived relations are defined similarly as for event structures except for \( \text{rs}, \text{release}, \text{sw}, \text{and hb} \) which are defined with \( \text{rf} \) instead of \( \text{jf} \):

\[
G \text{rs} \triangleq [G \text{W}; (G \text{po})_{\text{=1oc}}^2; (G \text{rf}; G \text{rmw})^* \\
G \text{release} \triangleq [G \text{E}_{\text{rel}}]; ([G \text{F}]; G \text{po})^2; G \text{rs} \\
G \text{sw} \triangleq G \text{release}; G \text{rf}; (G \text{po}; [G \text{F}])^2; [G \text{E}_{\text{acq}}^2] \\
G \text{hb} \triangleq (G \text{po} \cup G \text{sw})^+
\]

Following [12] we also define \( SC \text{-before} \) relation \( \text{scb} \) and \( \text{partial SC} \) relations \( \text{psc}_{\text{base}} \) and \( \text{psc}_{\text{F}} \).

\[
G \text{scb} \triangleq G \text{po} \cup G \text{po}_{\neq 1oc} \cup G \text{hb}; G \text{po}_{\neq 1oc} \cup G \text{hb} =_{1oc} \cup G \text{co} \cup G \text{fr}
\]
Next we show how to extract an execution graph from the event structure.

\textbf{Definition 3.} A set of events $X$ is called extracted from $S$ if the following conditions are met:

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

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 $X$:

- $G.E \equiv X$
- $G.tid \equiv S.tid | X$
- $G.lab \equiv S.lab | X$
- $G.po \equiv [X] ; S.po ; [X]$
- $G.rmw \equiv [X] ; S.rmw ; [X]$
- $G.rf \equiv [X] ; S.rf ; [X]$
- $G.co \equiv [X] ; S.co ; [X]$

We say that such execution graph $G$ is \textit{associated with} $X$ and that it is \textit{extracted} from the event structure: $S \triangleright G$.

Finally, we define the consistency of an execution graph. We say that only the consistent execution graphs, extracted from the \textit{Weakestmo} consistent event structure, constitute the set of program executions under \textit{Weakestmo} memory model.

\textbf{Definition 4.} Execution graph $G$ is \textit{Weakestmo}-consistent if the following hold:

- $G.\text{hb} ; G.\text{eco}'$ is irreflexive. \hspace{1cm} (COHERENCE)
- $G.\text{rmw} \cap (G.\text{fr} ; G.\text{co}) \subseteq \emptyset$. \hspace{1cm} (ATOMICITY)
- $G.\text{psc}_{\text{base}} \cup G.\text{psc}_F$ is acyclic. \hspace{1cm} (SC)

### 3.6 Event Structure Construction

The event structure is constructed operationally, in a way that guarantees $po \cup jf$ to be acyclic. Thus, the \textit{Weakestmo} model prevents an appearance of thin-air reads by construction. The operational semantics is relatively complicated and proceeds in several stages.

First, we assume there is a small step operational semantics $\sigma \xtag{es} \tau \sigma'$ defining the sequential execution of the thread with the identifier $t$. It is defined on the set of thread states $\sigma, \sigma' : \text{ThreadState}$. Thread state $\sigma$, among other components, contains list of instructions $\text{instrs}$, program counter $pc \in \mathbb{N}$ which points to the next instruction to be executed, and partial execution graph $G$, corresponding to the execution of the thread up to the current state. Partial execution graph
Reconciling Event Structures with Modern Multiprocessors

is organized simpler that the full execution graph. It contains only the events of the given thread and it does not record reads-from and coherence order relations \((\sigma.G.E \subseteq G.\text{thread}(t))\) and \(\sigma.G.rf = \sigma.G.co = \emptyset\). Step of thread sequential semantics is labeled by the list of events \(es\), which is either empty (in case when the thread performs some local actions, like a conditional jump), contains single event (in case of executing load, store or fence instructions), or a pair of events (when some read-modify-write instruction is executed).

Second, there is a relation \(S \xrightarrow{es} S'\), called basic step, defined on the event structures, which is mainly responsible for the update of the set of events \(E\), program order \(po\), and read-modify-write pairs \(rmw\). Internally, basic step performs several thread-local steps \(\sigma \rightarrow^* t\sigma'\), until one of this steps will produce non-empty list of events \(es\). These events are added to the event structure on the step (i.e., \(S'.E = S.E \cup es\)).

Next, there are three relations \(S \xrightarrow{(w,r)} S'\), \(S \xrightarrow{(W,w)} S'\) and \(S \xrightarrow{(W,W,w)} S'\), each of them is responsible for the update of the corresponding component of the event structure. The \(S \xrightarrow{(w,r)} S'\) relation is labeled by the write event \(w\) and read event \(r\), which form a new \(jf\) edge in the updated event structure \(S'\). The \(S \xrightarrow{(W,w)} S'\) is labeled by the set of writes \(W \subseteq S.w\) and a write event \(w\). The set \(W\) should be an \(S.ev\)-equivalence class, in the updated event structure the write \(w\) will be added to this class. Finally, the \(S \xrightarrow{(W,W,w)} S'\) is labeled by \(S.ev\)-equivalence class \(W\), the set of writes \(W \subseteq S.w\) and a write \(w\). The set \(W\) should be prefix-closed with respect to \(S.co\) and disjoint with \(W\). The event \(w\) will be placed \(S.co\)-after events from \(W\) in \(S'\) and before the \(S.co\)-complement of this set.\(^{16}\)

The relation \(S \xrightarrow{es} S'\) defines the whole transition relation using auxiliary relations mentioned above. It consists of four cases, which correspond to execution of fence, load, store or atomic update instructions. The relation \(S \xrightarrow{es} S'\) additionally ensures that \(S'\) is consistent.

Because of space constraints, we refer the reader to \([5]\) and our Coq developments for the full formal definitions related to the event structure construction.

### 4 IMM\(_{SC}\): IMM extended with SC accesses

Unlike \texttt{Weakestmo}, IMM tracks syntactic dependencies in its execution graphs, and uses them to forbid “out-of-thin-air” behaviors.

**Definition 5.** An IMM execution graph \(G\) is a tuple:

\[
\langle E, \text{tid}, \text{lab}, po, \text{rmw}, rf, \text{co}, \text{data}, \text{addr}, \text{ctrl}, \text{casdep} \rangle
\]

\(^{16}\) Since the \(co\) forms a total order on a set of writes with the same location, it is sufficient to pick just a single write event and place the event \(w\) \(co\)-after it. However, we found it technically more convenient to define the position by the set of write events.
where \( \langle E, tid, lab, po, rmw, rf, co \rangle \) is an execution graph (Def. 2), and data, addr, ctrl, casdep are relations that represent data, address, control and CAS dependencies respectively. They should satisfy the following constraints:

- \( \text{data} \subseteq \mathbb{R}; po : [w] \)
- \( \text{addr} \subseteq \mathbb{R}; po : [rw] \)
- \( \text{ctrl} \subseteq \mathbb{R}; po \)
- \( \text{ctrl} : po \subseteq \text{ctrl} \)
- \( \text{casdep} \subseteq \mathbb{R}; po : [\text{dom}(\text{rmw})] \)

IMM-consistency is defined as follows \[17\]:

**Definition 6.** An execution graph \( G \) is IMM-consistent if the following hold:

- \( G.\text{hb}; G.\text{eco}' \) is irreflexive. (COHERENCE)
- \( G.\text{rmw} \cap (G.\text{fr}; G.\text{co}) \subseteq \emptyset. \) (ATOMICITY)
- \( G.\text{ar} \) is acyclic. (NO-THIN-AIR)

Putting aside SC accesses, which are not included in IMM, this definition extends Weakestmo-consistency (Def. 4) with the NO-THIN-AIR constraint. The relation \( \text{ar} \) is defined as follows (eliding the “G.” prefix):\[17\]

\[
\begin{align*}
\text{bob} & \triangleq po : [w^{=\text{rel}}] \cup [w^{=\text{loc}}] ; po \cup [w^{=\text{acq}}]; po \cup [F] \cup [F] ; po \\
\text{deps} & \triangleq \text{data} \cup \text{ctrl} \cup \text{addr}; po^2 \cup \text{casdep} \cup [\text{dom}(\text{rmw})]; po \\
\text{ppo} & \triangleq \mathbb{R}; (\text{deps} \cup \text{rfe})^+ ; [w] \\
\text{detour} & \triangleq (\text{co}; \text{rfe}) \cap po \\
\text{ar} & \triangleq \text{rfe} \cup \text{bob} \cup \text{ppo} \cup \text{detour} \cup \text{psc}_F
\end{align*}
\]

To handle SC accesses, we define an extension of IMM, which we call IMM\(_{SC}\). Its consistency predicate is defined as follows:

**Definition 7.** An execution graph \( G \) is IMM\(_{SC}\)-consistent if, in addition to the conditions in Def. 6, the following hold:

- \( G.\text{psc}_{\text{base}} \cup G.\text{psc}_F \) is acyclic. (sc)

The SC constraint is taken as is from Weakestmo-consistency and RC11-consistency \[12\], Definition 1]. Since \( \text{psc}_F \) is already included in IMM’s \( \text{ar} \) relation, one may consider the natural option of including \( \text{psc}_{\text{base}} \) in \( \text{ar} \) as well. However, it leads to a too strong model, as it forbids the following behaviour:

\[
\begin{align*}
a & := [x]^{=\text{rli}}; [y]^{=\text{sc}} := 1 \quad \#2 \quad \| \quad [y]^{=\text{sc}} := 2 \quad \| \quad [x]^{=\text{rli}} := b \\quad \#2 \\
b & := [y]^{=\text{rli}} \quad \text{R}^{=\text{rli}}(x, 2) \quad \text{w}^{=\text{sc}}(y, 2) \quad \text{R}^{=\text{rli}}(y, 2)
\end{align*}
\]

This behaviour is allowed by POWER (using any of the two intended compilation schemes for SC accesses).

\[17\] In \[17\], \( \text{ar} \) also includes \( [w^{=\text{strong}}] ; po ; [w] \) which introduces ordering from some RMW write events. In PS to IMM compilation, it was required that all RMWs in a program are strong ones. There is no such restriction for Weakestmo to IMM\(_{SC}\) compilation, thus we do not distinguish strong RMWs.
4.1 Compiling IMM\textsubscript{SC} to Hardware

The main benefit of IMM is its use to simplify compilation correctness proofs by breaking them into two parts: (i) correctness of mapping from the high-level language to IMM; and (ii) correctness of mapping from IMM to the different multiprocessor architectures. In this section, we establish part (ii) for IMM\textsubscript{SC} by extending the results of [17] to support SC accesses with their intended compilation schemes to the different architectures.

As was done in [17], since IMM\textsubscript{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 IMM\textsubscript{SC} execution graphs to target architecture execution graphs that is induced by compilation of IMM\textsubscript{SC} programs to machine programs. Hence, roughly speaking, for each architecture $\alpha \in \{\text{TSO, POWER, ARMv7, ARMv8, RISC-V}\}$, our (mechanized) result takes the following form:

If the $\alpha$-execution-graph $G_{\alpha}$ corresponds to the IMM\textsubscript{SC}-execution-graph $G$, then $\alpha$-consistency of $G_{\alpha}$ implies IMM\textsubscript{SC}-consistency of $G$.

Since the mapping from Weakestmo to IMM\textsubscript{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 $\alpha$ that we consider. The exact notions of correspondence of $G_{\alpha}$ and $G$ is presented in Appendices A to C.

The mapping of IMM\textsubscript{SC} to each architecture, follows the intended compilation scheme of C/C++11 in [14,12], and extends the corresponding mappings of IMM from [17] with the mapping of SC reads and writes. Next, we schematically present these extensions.

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

| Fence after SC writes | Fence before SC reads |
|-----------------------|-----------------------|
| $([R^sc]) \triangleq \text{mov}$ | $([R^sc]) \triangleq \text{mfence;mov}$ |
| $([W^sc]) \triangleq \text{mov;mfence}$ | $([W^sc]) \triangleq \text{mov}$ |
| $([RMW^{sc}]) \triangleq (\text{lock}) \text{xchg}$ | $([RMW^{sc}]) \triangleq (\text{lock}) \text{xchg}$ |

The first, which is implemented in mainstream compilers, inserts an mfence after every SC write; which 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.

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

| Leading sync | Trailing sync |
|--------------|---------------|
| $([R^{sc}]) \triangleq \text{sync;}[R^{acq}]$ | $([R^{sc}]) \triangleq \text{ld;}\text{sync}$ |
| $([W^{sc}]) \triangleq \text{sync;}\text{st}$ | $([W^{sc}]) \triangleq ([W^{rel}])\text{;}\text{sync}$ |
| $([RMW^{sc}]) \triangleq \text{sync;}[RMW^{acq}]$ | $([RMW^{sc}]) \triangleq ([RMW^{rel}])\text{;}\text{sync}$ |
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 IMMSC. That is, this global execution graph transformation cannot make an inconsistent execution consistent:

**Theorem 2.** Let $G$ be an execution graph such that

$$[\text{RW}^{sc}]; (G,po' \cup G,po'; G,hb; G,po') \cup [\text{RW}^{sc}] \subseteq G,\text{hb} \cup [F^{sc}]; G,\text{hb},$$

where $G,po' \triangleq G,po \setminus G,\text{rmw}$. 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, IMMSC-consistency of $G$ follows from IMM-consistency of $G'$.

Having this theorem, we can think about mapping of IMMSC to POWER as if it consists of three steps. We establish the correctness of each of them separately.

1. At the IMMSC level, we globally split each SC-access to an SC-fence and release/acquire access. Correctness of this step follows by Theorem 2.
2. We map IMM to POWER, whose correctness follows by the existing results of [17], 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;1wsync;st for an SC write. The 1wsync is redundant here since sync provides stronger guarantees than 1wsync 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.

**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. [17] 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 |
|-------------|--------------|
| $([R^{sc}]) \triangleq \text{dmb};([R^{seq}])$ | $([R^{sc}]) \triangleq \text{ldr};\text{dmb}$ |
| $([W^{sc}]) \triangleq \text{dmb};\text{str}$ | $([W^{seq}]) \triangleq ([W^{rel}]);\text{dmb}$ |
| $([RMW^{sc}]) \triangleq \text{dmb};([RMW^{seq}])$ | $([RMW^{sc}]) \triangleq ([RMW^{rel}]);\text{dmb}$ |
ARMv8 The mapping to ARMv8 [19] is defined in a straightforward fashion, since ARMv8 has dedicated instructions for SC accesses:

\[
\begin{align*}
[R^{sc}] & \triangleq \text{ldar} \\
[W^{sc}] & \triangleq \text{stlr} \\
[FADD^{sc}] & \triangleq L:ldaxr;stlxr;bc L \\
[CAS^{sc}] & \triangleq L:ldaxr;cmp;bc Le;stlxr;bc L;Le:
\end{align*}
\]

RISC-V The RISC-V model [21,22] is stronger than the ARMv8 model. Therefore, soundness of mapping to RISC-V follows from soundness of mapping to ARMv8.

4.2 Compiling C11 and RC11 to IMM_SC

Podkopaev et al. [17] also proved the correctness of the mapping from RC11 without SC and non-atomic accesses to IMM. The extension of this result to cover SC accesses using IMM_SC is straightforward since IMM_SC has the same sc axiom as RC11. Similarly, the correctness of the mapping from C11, including SC and non-atomic accesses (after applying the fixes of [23] and [12]), to IMM_SC is trivial.

5 Simulation Relation for Weakestmo to IMM_SC Proof

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

The way we define \(\mathcal{I}(\text{prog}, G, (C, I), S, X)\) induces a strong connection between events in the execution graph \(G\) and the event structure \(S\). We make this connection explicit with the function \(s2g_{G,S} : S.E \rightarrow G.E\), which is defined in a way that satisfies the following predicate:\(^{18}\)

\[
\forall e \in S.E, S.tid(e) = G.tid(s2g_{G,S}(e)) \land \\
(e \in S.Init \Rightarrow S.loc(e) = G.loc(s2g_{G,S}(e))) \land \\
|\text{dom}(S.po ; [e])| = |\text{dom}(G.po ; [s2g_{G,S}(e)])|.
\]

That is, \(e\) and \(s2g_{G,S}(e)\) belong to the same thread and have the same po-position in the thread. Note that \(s2g_{G,S}\) does not have to be injective since if events \(e\) and \(e'\) are in immediate conflict in \(S\) they have the same \(s2g_{G,S}\)-image in \(G\).

\(^{18}\) 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.
In combination with $s2g_{G,S}$, we often use functorial $\llceil \cdot \rrceil_f$ and co-functorial $\llfloor \cdot \rrfloor_f$ maps for sets and relations:

$$\llceil f \rrceil_f : (f : A \to B) \to (S_A : 2^A) \to 2^B \triangleq \{ f(e) \mid e \in S_A \}$$

$$\llfloor f \rrfloor_f : (f : A \to B) \to (S_B : 2^B) \to 2^A \triangleq \{ e \mid f(e) \in S_B \}$$

$$\llceil R_A \rrceil_f : (f : A \to B) \to (R_A : 2^{A \times A}) \to 2^{B \times B} \triangleq \{ (f(e), f(e')) \mid (e, e') \in R_A \}$$

$$\llfloor R_B \rrfloor_f : (f : A \to B) \to (R_B : 2^{B \times B}) \to 2^{A \times A} \triangleq \{ (e, e') \mid (f(e), f(e')) \in R_B \}$$

For example, $\llceil C \rrceil_{s2g,G,S}$ denotes a subset of $S$'s events whose $s2g$-images are covered events in $G$, and $\llfloor S \rrfloor_{s2g,G,S}$ denotes a relation on events in $G$ whose $s2g$-preimages in $S$ are related by $S \cdot r mw$.

In the rest of the paper, we do not write the subscript for $s2g$ since the parameters for this function are usually deducible from the context. Moreover, we use the notation $\llceil \cdot \rrceil$ and $\llfloor \cdot \rrfloor$ only in combination with $s2g$ and we omit the subscript (i.e., set and rel) since it can be deduced from the context (e.g., we write just $\llceil S \rrceil_{s2g}$ instead of $\llceil [s2g] \rrceil$).

In the simulation relation $I$, we also use a function $S.K_C : \text{Tid} \to \text{ThreadState}$ parameterized by a covered set $C$:

$$S.K_C(t) \triangleq \begin{cases} S.K_{\text{init}}(t) & \text{if } X \cap S.\text{thread}(t) \cap [C] = \emptyset \\ S.K(e) & \text{if } e \text{ is } S.\text{po}-\text{last in } X \cap S.\text{thread}(t) \cap s2g[C] \end{cases}$$

Intuitively, this function returns the state of thread $t$’s local semantics that represents thread $t$’s covered part of the simulated execution $X$.

To define the simulation relation $I(prog, G, TC, S, X)$, we introduce an auxiliary relation, $I_T(prog, G, TC, S, X)$, parameterized by a set of threads $T$. The relation $I_T$ itself is defined to be $I_{\text{tid}}$ where $\text{Tid}$ is the complete set of threads. The auxiliary relation $I_T$ is used later in the proof of Lemma 2 in §6, where a new branch in the event structure has to be constructed via multiple steps. In the middle of these steps, the full simulation relation $I$ may be temporarily broken for the thread for which the branch is constructed.

We define the relation $I_T(prog, G, (C, I), S, X)$ to hold if the following conditions are met:

1. $G$ is an IMM$_{SC}$-consistent execution of $prog$.
2. $S$ is a Weakestmo-consistent event structure of $prog$.
3. $X$ is an extracted subset of $S$.
4. The $s2g$-image of $X$ is equal to the union of the covered and issued events and the events which po-precede the issued ones:
   $$\llceil X \rrceil \equiv C \cup dom(G . \text{po}^\forall : [I])$$
5. The $s2g$-image of $S$’s event has the same type, modifier, and location. Additionally, the $s2g$-image of $X$’s event which is covered or issued has the same value:
   (a) $\forall e \in S.E. S.(\text{typ, loc, mod})(e) = G.\{\text{typ, loc, mod}\}(s2g(e))$
Reconciling Event Structures with Modern Multiprocessors

6. The $s2g$-image of a justification edge is in $G.rf$ if the edge ends either in $X \cap \|C \cup I\|$, domain of $S.rmw$, an acquire access, or followed by an acquire fence:
   - $\|S.jf : [X \cap \|C\|]\| \subseteq G.rf$
   - $\|S.jf : S.rmw\| \subseteq G.rf : G.rmw$
   - $\|S.jf : (S.po ; [S.F]) ; [S.E^{acq}]\| \subseteq G.rf ; (G.po ; [S.F]) ; [G.E^{acq}]$

7. Each write event in $S$ which justifies some read event externally should be $S.ev$-equal to a write event in $X$ whose $s2g$-image is issued:
   - $dom(S.jfe) \subseteq dom(S.ev : [X \cap \|I\|])$

8. The $s2g$-image of $S.ev$ is a subset of the identity relation:
   - $\|S.ev\| \subseteq id$

9. Let $w$ and $w'$ be different events in one $S.ev$ equivalence class. Then, there is $w''$ in this equivalence class s.t. $w''$ is in $X$ and $s2g(w'')$ is issued:
   - $S.ev \subseteq (S.ev : [X \cap \|I\|] ; S.ev)''$

10. The $s2g$-image of $S.co$ lies in the reflexive closure of $G.co$. Additionally, $s2g$-images of $S.co$-edges ending in $X \cap S.thread(T)$ lay in $G.co$:
    - $\|S.co\| \subseteq G.co''$
    - $\|S.co : [X \cap S.thread(T)]\| \subseteq G.co$

11. The $s2g$-image of a justification edge is included in paths in $G$ representing observation of the corresponding thread:
    - $\|S.jf\| \subseteq G.rf'' : (G.hb ; [G.F^sc])'' ; G.psc'' ; G.hb''$

12. The $s2g$-image of $S.rmw$ is in $G.rmw$. Vice versa, $G.rmw$ ending in the covered set is in the $s2g$-image of $S.rmw$ ending in $X$.
    - $\|S.rmw\| \subseteq G.rmw$
    - $G.rmw ; [C] \subseteq \|S.rmw : [X]\|$

13. Let $e$, $w$, and $w'$ be events in $S$ s.t. (i) $(e, w)$ is an $S.release$ edge, (ii) $w$ and $w'$ is in the same $S.ev$ equivalence class, (iii) $w'$ is in $X$, and (iv) $s2g(w')$ is issued. Then $e$ is in $X$:
    - $dom(S.release : S.ev : [X \cap \|I\|]) \subseteq X$

This property is needed to show that $dom(S.hb \setminus S.po)$ is included in $X$.

14. Let $r$, $r'$, $w$, and $w'$ be events in $S$ s.t. (i) $r$ and $r'$ are in immediate conflict and justified from $w$ and $w'$ respectively, and (ii) $r'$ is in $X$ and its thread is in $T$. Then $s2g(w)$ is $G.co$-less than $s2g(w')$:
    - $\|S.jf : S.cf_{imm} : [X \cap S.thread(T)] ; S.jf^{-1}\| \subseteq G.co$

This property is needed to prove $cf_{imm}$-JUSTIFICATION on the simulation step.

15. For all $t \in T$ there exists $\sigma$ s.t. $S.K_C(t) \rightarrow^*_t \sigma$ and the thread-local execution graph $\sigma G$ is equivalent modulo $rf$ and $co$ components to the restriction of $G$ to the thread $t$. 

(b) $\forall e \in X \cap \|C \cup I\|$. $S.val(e) = G.val(s2g(e))$
6 Simulation Step Proof

In this section, we present an outline of our proof of Lemma 2, which states that the simulation relation $I$ can be restored after a traversal step.

Suppose that $I(prog, G, TC, S, X)$ holds for some $prog$, $G$, $TC$, $S$, and $X$, and we need to simulate a traversal step $G \vdash TC \rightarrow t TC'$ s.t. it either covers or issues an event in thread $t$. Then we need to provide an event structure $S'$ and a subset of its events $X'$ s.t. $I(prog, G, TC', S', X')$ holds. Weakestmo might need to take multiple steps from $S$ to $S'$, i.e., to construct a new so-called certification branch to have representatives of all issued write events from $TC'$.

For example, consider construction of the certification branch to simulate issuing of $e_{12}^{1}$: $W(y, 1)$, i.e., the traversal step

$$G_{LB} \vdash \langle G_{LB}.Init, G_{LB}.Init \rangle \rightarrow_{1} \langle G_{LB}.Init, G_{LB}.Init \cup \{e_{12}^{1}\} \rangle,$$

ending in Fig. 4a. Before the step,

$$I(LB, G_{LB}, \langle G_{LB}.Init, G_{LB}.Init \rangle, S_{init}(LB), S_{init}(LB).E)$$

holds. To simulate the step, we need to start by showing that it is possible to execute instructions of $LB$'s thread $I$, which are '$a := [x]; [y] := 1$', in a way that they would produce a sequence of labels satisfying two properties:

1. For each read label in the sequence, it would be possible to find either a related write event in $S_{init}(LB)$ or a related write label in the sequence.
2. The sequence would contain labels of all issued write events in $G.thread(1) \cap (G_{LB}.Init \cup \{e_{12}^{1}\})$.

In our case, the sequence is $[\mathcal{R}(x, 0), \mathcal{W}(y, 1)]$. The first requirement arises from the fact that, for all read events to be added to the certification branch, we need to have write events to justify from. The second one means that, regardless of changing values read by some instructions, it is still possible to write the same values to the same locations as in the issued set after the traversal step.

In the general case, to construct such a sequence, we follow the approach of [17] for a similar problem with certifying promises in the compilation proof from $PS$ to $IMM$. For each read event $r$ in the part of the graph for which we construct the certification branch (i.e., $r \in G.thread(t) \cap (C' \cup dom(G.po ; [I'])))$, we choose from which write event the certification version of $r$ will read from. In order to do that, we introduce several auxiliary definitions.

First, we define the set of determined events, which depends on the execution graph $G$ and traversal configuration $(C', I')$:

$$G.determined((C', I')) \triangleq C' \cup I' \cup dom(G.rfi^2 : G.ppo ; [I']) \cup codom([I'] : G.rfi) \cup codom(G.rfe : [G.E^{\text{acq}}])$$

Intuitively, $G.determined((C', I'))$ represents the events which should be equal to their counterparts in the certification branch. In particular, it means that
these events should have the same label in graph $G$ and in the certification branch, and additionally the read events should have the same reads-from source. Note that set of determined events contain $\text{dom}(G, \text{rf})^2$; $G, \text{ppo} : \left[ I' \right]$—a set of events whose values the issued write events depend on, and $\text{codom}(G, \text{fe})$; $\left[ G, E_{\text{asq}} \right]$—a set of read events with mode equal or stronger than $\text{acq}$ that read value from another thread.

Second, we introduce the viewfront relation:

$$G.\text{vf}((C', I')) \triangleq [G, \text{w}] ; (G, \text{rf} ; [C'])^2 ; G, \text{hb}^2 ; G, \text{psc}^2 \cup G, \text{vf} ; [G, \text{determined}((C', I'))] ; G, \text{po}^2$$

For any event $e \in G, E$ the set $\text{dom}(G, \text{vf}((C', I')) ; [e])$ contains write events that are ‘observable’ by $e$.

Finally, we construct the simulation read from relation denoted $\text{sim}\_\text{rf}$:

$$G.\text{sim}\_\text{rf}((C', I')) \triangleq G.\text{vf}((C', I'))|_{G, \text{loc}} \setminus (G, \text{co} ; G, \text{vf}((C', I')))$$

It relates a read event $r$ to the $\text{co}$-last ‘observable’ write event with same location. Assuming that $G$ is $\text{IMM}_{\text{SC}}$-consistent, it can be shown that $G.\text{sim}\_\text{rf}$ agrees with $G, \text{rf}$ for the determined reads.

$$G.\text{sim}\_\text{rf}((C', I')) ; [G, \text{determined}((C', I'))] \subseteq G, \text{rf}$$

Having $\text{sim}\_\text{rf}$ as a guide for values read by instructions in the certification run, we construct the steps of the thread-local operational semantics

$$\sigma \rightarrow^*_t \sigma'$$

with the required sequence of labels using the receptiveness property\(^\text{19}\) of the thread’s semantics. Here $\sigma$ is a thread state corresponding to the $\text{S,po}$-last covered event from $X$ with thread identifier $t$, that is $\sigma = S, K_C(t)$ and $\sigma, G, E$ is equal to $G, \text{thread}(t) \cap C$.

Then, the proof of Lemma 2 is done by induction on $\sigma \rightarrow^*_t \sigma'$ using an auxiliary relation $\text{I}^{\text{cert}}$. Formally, this is stated in the following lemmas.

**Lemma 4.** Suppose $\text{I}(\text{prog}, G, T, C, S, X)$ and $G \vdash T \rightarrow^t T, C'$. Let $\sigma = S, K_C(t)$ and $\text{Br}_0 = X \cap (\text{S,init} \cup S, \text{thread}(t)) \cap \llfloor C \rrfloor$. Then there exists $\sigma'$ s.t. $\sigma \rightarrow^*_t \sigma'$ is a certification branch and $\text{I}^{\text{cert}}(\text{prog}, G, T, C', S, X, t, \text{Br}_0, \sigma, \sigma')$ holds.

**Lemma 5.** If $\text{I}^{\text{cert}}(\text{prog}, G, T, C', S, X, t, \text{Br}, \sigma, \sigma'')$ holds and $\sigma \neq \sigma''$, then there exist $\sigma'$ and $S'$ s.t. $\sigma \rightarrow^*_t \sigma'$, $\sigma' \rightarrow^*_t S'$, and $\text{I}^{\text{cert}}(\text{prog}, G, T, C', S', X, t, \text{Br} \cup S, \sigma', \sigma'')$ hold.

**Lemma 6.** If $\text{I}^{\text{cert}}(\text{prog}, G, T, C', S, X, t, \text{Br}, \sigma, \sigma')$ and $\sigma = \sigma'$ hold then $\text{I}(\text{prog}, G, T, C', S, X \setminus S, \text{thread}(t) \cup \text{Br})$ holds.

---

\(^{19}\) The definition of the receptiveness property is quite elaborate and technical. For the detailed definition we refer the reader to the Coq development of $\text{IMM}$ [6].
constructs $\sigma \rightarrow_t^* \sigma'$ and serves as the base case of induction. Lemma 5 is the induction step. Lemma 6 concludes the proof of Lemma 2 by restoring the simulation relation $\mathcal{T}$.

We define the relation $\mathcal{I}^{\text{cert}}(\text{prog}, G, \langle C, I \rangle, \langle C', I' \rangle, S, X, t, Br, \sigma, \sigma')$, where the parameter $Br$ represents the already constructed part of the certification branch, to hold if the following conditions are met:

1. $\mathcal{I}_{T \setminus \{t\}}(\text{prog}, G, \langle C, I \rangle, S, X)$ holds.
2. $G \vdash (C, I) \rightarrow_t (C', I')$ holds.
3. $\sigma$ and $\sigma'$ are thread states s.t. $\sigma$ corresponds to the $S, \text{po}$-last event in $Br$ and $\sigma'$ is reachable from $\sigma$, i.e., $\sigma \rightarrow_t^* \sigma'$.
4. The partial execution graph of $\sigma'$ contains covered and issued events up to the $G, \text{po}$-last issued write in thread $t$:
   - $\sigma'.G.E \equiv G.t\text{hread}(t) \cap (C' \cup \text{dom}(G.t\text{hread}(\sigma').[I']))$
5. The partial execution graph of $\sigma'$ assigns same type, location and mode as the full execution graph $G$ does. Additionally, it assigns the same value as $G$ to determined events.
   (a) $\forall e \in \sigma'.G.E. \sigma'.G.t\{\text{typ}, \text{loc}, \text{mod}\}(e) = G.t\{\text{typ}, \text{loc}, \text{mod}\}(e)$
   (b) $\forall e \in \sigma'.G.E \cap G.determined((C', I'))$. $\sigma'.G.val(e) = G.val(e)$
6. The set $Br$ consists of initial events plus the events from the thread $t$ and covered prefixes of $Br$ and $X$ restricted to thread $t$ coincide:
   (a) $Br \subseteq S, \text{init} \cup S.t\text{hread}(t)$
   (b) $Br \cap \|C\| \equiv X \cap (S, \text{init} \cup S.t\text{hread}(t)) \cap \|C\|
7. The $s2g$-image of the $jf$ relation ending in $Br$ is included in $G.jf_{\langle C', I' \rangle}$:
   - $\|S.jf; [Br]\| \subseteq G.jf_{\langle C', I' \rangle}$
8. For every issued event from $Br$ there exists an $S, \text{ev}$-equivalent in $X$. And, symmetrically, every issued event from $X$ within the processed part of the certification branch has an $S, \text{ev}$-equivalent in $Br$.
   (a) $Br \cap \|I\| \subseteq \text{dom}(S.ev; [X])$
   (b) $X \cap (I \cap \sigma.G.E) \subseteq \text{dom}(S.ev; [Br])$
9. The $s2g$-image of $S, \text{co}$ ending in $Br$ lies in $G, \text{co}$.
   The $s2g$-image of $S, \text{co}$ ending in $X \cap S.t\text{hread}(t)$ and not in the processed part of the certification branch lies in $G, \text{co}$.
   (a) $\|S, \text{co}; [Br]\| \subseteq G, \text{co}$
   (b) $\|S, \text{co}; [X \cap S.t\text{hread}(t) \setminus \|\sigma.G.E]\| \subseteq G, \text{co}$
10. Each $G, \text{rmw}$ edge ending in the processed part of the certification branch is the $s2g$-image of some $S, \text{rmw}$ edge ending in $Br$.
    - $G, \text{rmw}; [C' \cap \sigma.G.E] \subseteq \|S, \text{rmw}; [Br]\|$
11. Suppose $w, w', r$, and $r'$ are $S$’s events s.t. (i) $r$ and $r'$ are justified from $w$ and $w'$ respectively, and (ii) $r$ and $r'$ are in immediate conflict and belong to thread $t$. Then $s2g(w')$ is $G, \text{co}$-greater than $s2g(w)$ if either $r'$ is in $Br$:
    - $\|S.jf; S, \text{cfim}; [Br]; S, \text{cfim}^{-1}\| \subseteq G, \text{co}$
    or $r$ is not in $Br$ and $r'$ is in $X \cap S.t\text{hread}(t)$:
    - $\|S.jf; [S.E \setminus Br]; S, \text{cfim}; [X \cap S.t\text{hread}(t)]; S, \text{cfim}^{-1}\| \subseteq G, \text{co}$
Let us again consider the program LB from §2 together with its $\text{IMM}_{\text{SC}}$ execution graph depicted on Fig. 2a. We showcase the construction of the event structure following the traversal of Fig. 4 once again, but this time paying more attention to some particular properties preserved by $\mathcal{I}$ and $\mathcal{I}_{\text{cert}}$.

Initially, $\text{TC}_{\text{init}}(G_{\text{LB}}) \triangleq (G_{\text{LB}.\text{init}}, G_{\text{Init}})$, and $X_{\text{init}} \triangleq S_{\text{init}}(\text{prog}).\text{init}$ (it is easy to see that init forms a valid extracted subset, since all the constraints of Def. 3 are met). Since for $S_{\text{init}}(\text{prog})$ we have $\text{rmw} = \text{jf} = \text{co} = \emptyset$ and $\text{ew} = [w]; \text{id}; [\overline{w}]$ most of the properties of $\mathcal{I}$ hold trivially.

At the first step of the traversal $\text{TC}_{\alpha}$ the event $e_{1,1}^1: \mathcal{W}(y, 1)$ is issued. Steps Fig. 3a-Fig. 3b build the corresponding certification branch. The $G_{\text{sim.rf}}(\text{TC}_{\alpha})$ relation connects the read event $e_{1,1}^1$ with the initial write event $G_{\text{init}}$ since it is the only ‘observable’ event for this read. Thus on the step Fig. 3a the corresponding write event $S_{\text{init}}$ is chosen to justify the read event $e_{1,1}^1$ (note that $s_{2g}(S_{\text{init}}.\text{init}) = G_{\text{LB}.\text{init}}, s_{2g}(e_{1,1}^1) = e_{1,1}^1$ and consequently $[S_{\text{init}}.\text{jf}] \subseteq G_{\text{LB}.\text{sim.rf}}(\text{TC}_{\alpha})$). At the step Fig. 3b the certification branch $B_{\text{TS}} \triangleq \{s_{\text{init}}, e_{1,1}^1, e_{1,1}^2\}$ is fully constructed. It constitutes valid extracted subset of $S_{\text{b}}$ (denoted as $X_{\text{b}}$) since it has no conflicting events, $\text{rf}$-complete, contains only visible events, and $\text{hb}$-downward-closed.

Let us check that $\mathcal{I}(\text{prog}, G_{\text{LB}}, \mathcal{TC}_{\alpha}, X_{\text{b}}, X_{\text{b}})$ indeed holds. Given that $s_{2g}(S_{\text{init}}.\text{init}) = G_{\text{init}}, s_{2g}(e_{1,1}^1) = e_{1,1}^1, s_{2g}(e_{1,1}^2) = e_{1,1}^2$ it is easy to see that properties 4 and 5 hold. Properties 6a and 7 hold trivially since there are no covered reads nor external justification edges. Properties 8 and 9 also hold because $S_{\text{b}}$ contains only reflexive $\text{ew}$ edges. There is single $\text{co}$ edge in $S_{\text{b}}$: \{s_{\text{init}}, e_{1,1}^1\} $\in S_{\text{b}.\text{co}}$. Its $s_{2g}$-image lies in $G_{\text{co}}$: \{G_{\text{init}}, e_{1,1}^1\} $\in G_{\text{co}}$. Thus 10a and 10b also hold.

At the next step of traversal $\text{TC}_{\alpha}$ another write event $e_{1,2}^2: \mathcal{W}(x, 1)$ is issued. Steps Fig. 3c-Fig. 3d build the certification branch similarly as at the previous step. The only difference is that this time the read event $e_{1,2}^2$ is determined (since $\{e_{1,1}^2, e_{1,2}^2\} \in G_{\text{LB}.\text{ppo}}(I_{\text{b}})$). Because of that the simulation read-from relation $G_{\text{LB}.\text{sim.rf}}(\text{TC}_{\alpha})$ pick the write event $e_{1,2}^2$ as a source for $e_{1,2}^1$. The corresponding write event in $S_{\alpha} = e_{1,2}^1$, this write belongs to $X_{\text{b}}$ and its $s_{2g}$ image is issued therefore the constraint 7 is satisfied.

The next traversal step $\text{TC}_{\alpha}$ is challenging. The read event $e_{1,1}^1: \mathcal{R}(x, 1)$ is covered. Since the event structure $S_{\alpha}$ does not contain read events with $s_{2g}$ image equal to $e_{1,1}^1$ and label equal to $\mathcal{R}(x, 1)$, it has to ‘re-certify’ execution of the thread $I$. The new certification branch is constructed during the steps Fig. 3c-Fig. 3f. Notice at this point of the traversal the event $e_{1,1}^1$ becomes determined and thus $\{e_{2,1}^2, e_{1,1}^1\} \in G_{\text{LB}.\text{sim.rf}}(\text{TC}_{\alpha})$. It means that in the event structure the write event $e_{2,1}^2$ should be chosen to justify read $e_{1,1}^1$.

Note that after the construction of certification branch ends, it is not possible to just add events $e_{1,2}^1$ and $e_{2,2}^1$ to the extracted subset, since they are in conflict with the events $e_{1,1}^1$ and $e_{2,1}^1$. Thus in order to form new extracted subset $X_{\text{f}}$ we have to replace the later events with the former.

Notice the property 6a holds, because the only read event in $X_{\text{f}}$ with $s_{2g}$ image covered is $e_{1,2}^1$. Since its $s_{2g}$ image $e_{1,2}^1$ belongs to the set of determined events the $G_{\text{LB}.\text{sim.rf}}(\text{TC}_{\alpha})$ and $G_{\text{LB}.\text{rf}}$ assign the same write to it. The property 7 also holds, because $e_{2,1}^1$ belongs to $X_{\text{f}}$ and its $s_{2g}$ image is issued.
The event structure $S_f$ has single non-reflexive $\text{ew}$ edge: $\langle e_{21}^1, e_{22}^1 \rangle \in S_f.\text{ew}$. These two events form an $\text{ew}$ equivalence class. Both of them have same $s_2g$ image — event $e_{12}^1$, which is issued, and moreover the event $e_{22}^1$ belongs to $X_f$. Thus 8 and 9 hold. Notice that the $S_f.\text{ew}$ edge $\langle e_{21}^1, e_{22}^1 \rangle$ induces an $S_f.\text{rf}$ edge $\langle e_{22}^1, e_{12}^1 \rangle$, therefore the $\text{rf}$-completness constraint is satisfied for $X_f$. Also consider the $S_f.\text{cf} \cap (S_f.\text{je} \cup S_f.\text{jf})^*$ path between the events $e_{21}^1$ and $e_{12}^1$. Without the edge $\langle e_{21}^1, e_{22}^1 \rangle \in S_f.\text{ew}$ the existence of this path would make the event $e_{12}^1$ invisible, thus violating $X_f \subseteq S_f.\text{Vis}$ constraint.

Since $S_f$ do not have any new $\text{co}$ edges properties $10a$ and $10b$ still hold.

On the traversal step $TC_d$ event $e_{12}^1$: $\forall(y, 1)$ is covered. However the event structure $S_d$ is not updated since it already ‘fits’ the new traversal configuration. The same applies to traversal steps $TC_e$ and $TC_f$.

7 Related Work and Conclusion

While there are several memory model definitions both for hardware architectures [1,15,9,19,20] and programming languages [2,3,16,18,13,10] in the literature, there are relatively few compilation correctness results [11,12,24,8,17].

As a way to show correctness of Weakestmo compilation to hardware, we employed IMM, which we extended with SC accesses, from which compilation to hardware follows. The only limitation of this approach is that IMM enforces ordering between RMW events and subsequent memory accesses. This ordering is also typically enforced in hardware architectures (x86-TSO, POWER, ARMv7), but not always. The exception is ARMv8, which has two ways of compiling RMWs: one is via a pair of load-linked and store-conditional (LDX/STX) instructions in a loop, which naturally induces a dependency from RMWs to subsequent accesses, and the other is via hardware instructions, such as CAS, LDADD, LDCLR, LDMAX, LDMIN, which do not necessarily induce dependencies to subsequent instructions.

An alternative for compiling to ARMv8 would have been to use the recently developed Promising-ARM model [20]. Indeed, since Promising-ARM is closely related to PS [11], 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 [5]. 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. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simula-
tion, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. 36(2), 71–7:74 (Jul 2014). https://doi.org/10.1145/2627752, http://doi.acm.org/10.1145/2627752
2. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL 2011. pp. 55–66. ACM, New York (2011). https://doi.org/10.1145/1925844.1926394

3. Bender, J., Palsberg, J.: A formalization of java’s concurrent access modes. Proc. ACM Program. Lang. 3(OOPSLA), 142:1–142:28 (Oct 2019). https://doi.org/10.1145/3360568, http://doi.acm.org/10.1145/3360568

4. Boehm, H.J., Densky, B.: Outlawing ghosts: Avoiding out-of-thin-air results. In: MSCP 2014. pp. 7:1–7:6. ACM (2014). https://doi.org/10.1145/2618128.2618134

5. Chakraborty, S., Vafeiadis, V.: Grounding thin-air reads with event structures. Proc. ACM Program. Lang. 3(POPL), 70:1–70:27 (2019). https://doi.org/10.1145/3290383

6. The Coq development of IMM, available at http://github.com/weakmemory/imm (2019)

7. Deacon, W.: The ARMv8 application level memory model (2017), https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat

8. Dolan, S., Sivaramakrishnan, K., Madhavapeddy, A.: Bounding data races in space and time. In: PLDI 2018. pp. 242–255. ACM, New York (2018). https://doi.org/10.1145/3192366.3192421, http://doi.acm.org/10.1145/3192366.3192421

9. Flur, S., Gray, K.E., Pulte, C., Sarkar, S., Sezgin, A., Maranget, L., Deacon, W., Sewell, P.: Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In: POPL 2016. pp. 608–621. ACM, New York (2016). https://doi.org/10.1145/2837614.2837615, http://doi.acm.org/10.1145/2837614.2837615

10. Jeffrey, A., Riely, J.: On thin air reads towards an event structures model of relaxed memory. In: LICS 2016. pp. 759–767. ACM, New York (2016). https://doi.org/10.1145/2933575.2934536, http://doi.acm.org/10.1145/2933575.2934536

11. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: POPL 2017. pp. 175–189. ACM, New York (2017). https://doi.org/10.1145/3009837.3009850

12. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in C/++11. In: PLDI 2017. pp. 618–632. ACM, New York (2017). https://doi.org/10.1145/3062341.3062352

13. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL 2005. pp. 378–391. ACM, New York (2005). https://doi.org/10.1145/1040305.1040336

14. C/++11 mappings to processors (2016), http://www.cl.cam.ac.uk/~pes20/cpp/cpp0x mappings.html

15. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-DSO. In: TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009)

16. Pichon-Pharabod, J., Sewell, P.: A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In: POPL 2016. pp. 622–633. ACM, New York (2016). https://doi.org/10.1145/2837614.2837616

17. Podkopaev, A., Lahav, O., Vafeiadis, V.: Bridging the gap between programming languages and hardware weak memory models. Proc. ACM Program. Lang. 3(POPL), 69:1–69:31 (2019). https://doi.org/10.1145/3290382

18. Podkopaev, A., Sergey, I., Nanevski, A.: Operational aspects of C/++ concurrency. CoRR abs/1606.01400 (2016), http://arxiv.org/abs/1606.01400

19. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. Proc. ACM Program. Lang. 2(POPL), 19:1–19:29 (2018). https://doi.org/10.1145/3158107
20. Pulte, C., Pichon-Pharabod, J., Kang, J., Lee, S.H., Hur, C.K.: 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. pp. 1–15. PLDI 2019, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314624, http://doi.acm.org/10.1145/3314221.3314624
21. 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]
22. RISC-V: herd vs. operational models (2018), http://diy.inria.fr/cats7/riscv/
23. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Zappa Nardelli, F.: Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In: POPL 2015. pp. 209–220. ACM, New York (2015). https://doi.org/10.1145/2676726.2676995
24. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: ComCertTSO: A verified compiler for relaxed-memory concurrency. J. ACM 60(3), 22 (2013). https://doi.org/10.1145/2487241.2487248
From \textsc{IMM} to \textsc{ARMv8}

The intended mapping of \textsc{IMM} to \textsc{ARMv8} is presented schematically in Fig. 5 and follows [14]. Note that acquire and SC loads are compiled to the same instruction (\texttt{ldar}) as well as release and SC stores (\texttt{stlr}). In \textsc{ARM} assembly RMWs are represented as pairs of instructions—exclusive load (\texttt{ldxr}) followed by exclusive store (\texttt{stxr}), and these instructions are also have their stronger (SC) counterparts—\texttt{ldaxr} and \texttt{stlxr}.

We use \textsc{ARMv8} declarative model [7] (see also [19]). Its labels are given by:

- \textsc{ARM} read label: \( R^{\text{r}}(x, v) \) where \( x \in \text{Loc}, v \in \text{Val}, o_{R} \in \{rlx, Q, A\} \), and \( rlx \sqsubseteq Q \sqsubseteq A \).
- \textsc{ARM} write label: \( W^{\text{w}}(x, v) \) where \( x \in \text{Loc}, v \in \text{Val}, o_{W} \in \{rlx, L\} \), and \( rlx \sqsubseteq L \).
- \textsc{ARM} fence label: \( F^{\text{f}} \) where \( o_{F} \in \{ld, sy\} \) and \( 1d \sqsubseteq sy \).

In turn, \textsc{ARM}’s execution graphs are defined as \textsc{IMM}’s ones, except for the CAS dependency, \texttt{casdep}, which is not present in \textsc{ARM} executions.

The definition of \textsc{ARMv8}-consistency requires the following derived relations (see [19] for further explanations and details):

\[
\begin{align*}
\text{obs} & \triangleq \text{rfe} \cup \text{fre} \cup \text{coe} & \text{(observed-by)} \\
\text{dob} & \triangleq (\text{addr} \cup \text{data}) \setminus \text{rfi} \cup (\text{ctrl} \cup \text{data}) \setminus \text{w} \cup \text{coi} \cup \text{addr} \cup \text{po} \cup \text{w} & \text{(dependency-ordered-before)} \\
\text{aob} & \triangleq \text{rmw} \cup \text{w} \cup \text{fri} \cup [R^\triangledown] & \text{(atomic-ordered-before)} \\
\text{bob} & \triangleq \text{po} \cup [F^\triangledown] \cup \text{po} \cup \text{R} \cup [F^\triangledown] \cup \text{po} \cup \text{po} \cup [w^p] \cup [w^b] \cup [r^a] & \text{(barrier-ordered-before)}
\end{align*}
\]

\textbf{Definition 8.} An \textsc{ARMv8} execution graph \( G_{a} \) is called \textsc{ARMv8}-consistent if the following hold:

- \( \text{codom}(G_{a}, \text{rf}) = G_{a}, R \).
- For every location \( x \in \text{Loc}, G_{a}, \text{co} \) totally orders \( G_{a}, w(x) \).

\footnote{We only describe the fragment of the model that is needed for mapping of \textsc{IMM}, thus excluding isb fences.}
Theorem 3. Let \( G \) be an IMM execution graph with whole serial numbers (\( \mathbb{SN}[G.E] \subseteq \mathbb{N} \)), and let \( G_a \) be an ARMv8 execution graph that corresponds to \( G \). Then, ARMv8-consistency of \( G_a \) implies IMM\(_{\text{SC}}\)-consistency of \( G \).

Proof (Outline). IMM-consistency of \( G \) follows from [17, Theorem 4.5]. That is, we only need to show that the SC axiom holds for \( G \). We start by showing that \( G_a.\text{obs}' \cup G_a.\text{dob} \cup G_a.\text{aob} \cup G_a.\text{bob}' \) is acyclic, where

\[
\begin{align*}
\text{obs}' & \triangleq \text{rfe} \cup \text{fr} \cup \text{co} \\
\text{bob}' & \triangleq \text{bob} \cup \text{R}; \text{po}; [\text{F}^\text{ld}] \cup \text{po}; [\text{F}^\text{sy}] \cup [\text{F}^\text{lds}] \cup \text{po}
\end{align*}
\]

Then, we finish the proof by showing that \( G_a.\text{psc}_{\text{base}} \cup G_a.\text{psc}_F \) is included in \((G_a.\text{obs}' \cup G_a.\text{dob} \cup G_a.\text{aob} \cup G_a.\text{bob}')^+\). \( \square \)
That is, in this section, we concentrate only on the compilation alternative which

There are two possible alternatives for compiling SC accesses (see the bottom of

the compilation on execution graphs.

TSO

In turn,

N

labels are given by:

Fig. 6: Compilation scheme from IMM_SC to TSO.

G

B  From IMM_SC to TSO

The intended mapping of IMM_SC to TSO is presented schematically in Fig. 6. There are two possible alternatives for compiling SC accesses (see the bottom of Fig. 6): to compile an SC store to a store followed by a fence or to compile an SC load to a load preceded by a fence. Both of the schemes guarantee that in compiled code there is a fence between every store and load instructions originated from SC accesses. Regarding compilation schemes of SC accesses, our proof of compilation correctness from IMM_SC to TSO depends only on this property. That is, in this section, we concentrate only on the compilation alternative which compiles SC stores using fences.

As a model of the TSO architecture, we use a declarative model from [1]. Its labels are given by:

— TSO read label: R(x, v) where x ∈ Loc and v ∈ Val.
— TSO write label: W(x, v) where x ∈ Loc and v ∈ Val.
— TSO fence label: MFENCE.

In turn, TSO’s execution graphs are defined as IMM_SC’s ones. Below, we interpret the compilation on execution graphs.

Definition 10. Let G be an IMM execution graph with whole identifiers (G.E ⊆ N). A TSO execution graph G_t corresponds to G if the following hold:

— G_t.E = G.E \ G.F^\text{SC} \cup \{n + 0.1 \mid n \in G.W^\text{SC}\}
  (non-SC fences are removed)
— G_t.tid(e) = G.tid([e] + 0.1) for all e in G_t
— G_t.po =
  [G.tid(e); \{G.po \cup \{(a, n + 0.1) \mid (a, n) \in G.po^？\} \cup \{(n + 0.1, a) \mid (n, a) \in G.po\}];
  G_t.E]
  (new events are added after SC writes)
— G_t.lab = \{e \mapsto \{G.lab(e)\} \mid e \in G.E \setminus G.F^\text{SC}\} \cup \{e \mapsto MFENCE \mid e \in G_t.E \setminus G.E\}
  where:
  \(\langle R^\text{SC}(x, v)\rangle \triangleq R(x, v)\)
  \(\langle W^\text{SC}(x, v)\rangle \triangleq W(x, v)\)
  \(\langle F^\text{SC}\rangle \triangleq MFENCE\)
— G_rmw = G_t.rmw, G.data = G_t.data, and G.addr = G_t.addr
  (the compilation does not change RMW pairs and data/address dependencies)
– \( G.\text{ctrl}; [G.E \setminus G.F^\text{sc}] \subseteq G_t.\text{ctrl} \)

(the compilation only adds control dependencies)

The following derived relations are used to define the TSO-consistency predicate.

\[
\begin{align*}
\text{ppo}^{TSO} & \triangleq [R W]; \text{po}; [R W]; \text{po}; [R] \\
\text{fence}^{TSO} & \triangleq [R W]; \text{po}; [\text{MFENCE}]; \text{po}; [R W] \\
\text{implied\_fence}^{TSO} & \triangleq [W]; \text{po}; [\text{dom}(\text{rmw})] \cup [\text{codom}(\text{rmw})]; \text{po}; [R] \\
\text{hb}^{TSO} & \triangleq \text{ppo}^{TSO} \cup \text{fence}^{TSO} \cup \text{implied\_fence}^{TSO} \cup \text{rfe} \cup \text{co} \cup \text{fr}
\end{align*}
\]

**Definition 11.** \( G \) is called TSO-consistent if the following hold:

– \( \text{codom}(G.\text{rf}) = G.R \). \hspace{1cm} (\text{rf\text{-}COMPLETENESS})
– For every location \( x \in \text{Loc}, G.\text{co} \) totally orders \( G.W(x) \). \hspace{1cm} (\text{co\text{-}TOTALITY})
– \( \text{po}_{\text{loc}} \cup \text{rf} \cup \text{fr} \cup \text{co} \) is acyclic. \hspace{1cm} (\text{SC\text{-}PER\text{-}LOC})
– \( G.\text{rmw} \cap (G.\text{fre} ; G.\text{coe}) = \emptyset \). \hspace{1cm} (\text{ATOMICITY})
– \( G.\text{hb}^{TSO} \) is acyclic. \hspace{1cm} (\text{TSO\text{-}NO\text{-}THIN\text{-}AIR})

Next, we state our theorem that ensures IMM\text{SC}-consistency if the corresponding TSO execution graph is TSO-consistent.

**Theorem 4.** Let \( G \) be an IMM\text{SC} execution graph with whole identifiers \( G.E \subseteq \mathbb{N} \), and let \( G_t \) be an TSO execution graph that corresponds to \( G \). Then, TSO-consistency of \( G_t \) implies IMM\text{SC}-consistency of \( G \).

**Proof (Outline).** Since \( G_t \) corresponds to \( G \), we know that

\[
[G.W^\text{sc}]; G.\text{po}; [G.R^\text{sc}] \subseteq G_t.\text{po}; [G_t.\text{MFENCE}]; G_t.\text{po}
\]

as the aforementioned property of the compilation scheme. We show that

\[
G_t.\text{ehb}^{TSO} \triangleq G_t.\text{hb}^{TSO} \cup [G_t.\text{MFENCE}]; G_t.\text{po} \cup [G_t.\text{MFENCE}]; G_t.\text{po}
\]

is acyclic. Then, we show that \( G.\text{psc}_{\text{base}} \cup G.\text{psc}_F \) is included in \( G_t.\text{ehb}^{+} \). It means that the \text{sc} axiom holds for \( G \), and it leaves us to prove that \( G \) is IMM-consistent. That is done by standard relational techniques (see [6]). \( \square \)
Trailing \(14 \cup \{17\} \); \([\ldots] \) with elimination of the aforementioned redundancy of SC write compilation schemes [\ldots] (see Fig. 7) as in [17] for all instructions except for SC accesses. For the latter, there are two standard compilations schemes [14] presented in the bottom of Fig. 7: with leading and trailing sync fences.

The next definition presents the correspondence between IMM execution graphs and their mapped POWER ones following the leading compilation scheme in Fig. 7 with elimination of the aforementioned redundancy of SC write compilation.

**Definition 12.** Let \( G \) be an IMM execution graph with whole identifiers \((G.E \subseteq \mathbb{N})\). A POWER execution graph \( G_p \) corresponds to \( G \) if the following hold:

- \( G_p.E = G.E \cup \{ n + 0.1 \mid n \in (G.R_{\text{acq}} \cup \text{dom}(G.rmw)) \cup \text{codom}((G.R_{\text{acq}} ; G.rmw)) \) \cup \{ n - 0.1 \mid n \in (G.E_{\text{rel}} \cup \text{dom}(G.rmw)) \cup \text{dom}(G.rmw ; [G.W^{\text{rel}}])) \}

(new events are added after acquire reads and acquire RMW pairs and before SC accesses and SC RMW pairs)

- \( G_p.\text{tid}(e) = G.\text{tid}([e+0.1]) \) for all \( e \) in \( G_p \)

- \( G_p.\text{po} = G.\text{po} \cup \{(G.p.E \times G.p.E) \cap \{(a, n - 0.1) \mid \langle a, n \rangle \in G.\text{po}\} \cup \{(n - 0.1, a) \mid \langle n, a \rangle \in G.\text{po}^\#\} \cup \{(n, a + 0.1) \mid \langle a, n \rangle \in G.\text{po}^\#\} \cup \{(n + 0.1, a) \mid \langle a, n \rangle \in G.\text{po}\})\)

- \( G_p.\text{lab} = \{ e \mapsto \langle G.\text{lab}(e) \rangle \mid e \in G.E \} \cup \{ n + 0.1 \mapsto F_{\text{cross sync}} \mid n + 0.1 \in G_p.E \land n \in \mathbb{N}\} \cup \{ n - 0.1 \mapsto F_{\text{cross sync}} \mid n - 0.1 \in G_p.E \land n \in \mathbb{N}\land n \notin G.E_{\text{acq}} \cup \text{dom}(G.rmw ; [G.W^{\text{acq}}])\} \cup \{ n - 0.1 \mapsto F_{\text{sync}} \mid n - 0.1 \in G_p.E \land n \in \mathbb{N}\land n \notin G.E_{\text{sync}} \cup \text{dom}(G.rmw ; [G.W^{\text{sync}}])\} \)

where:

\[
\begin{align*}
\{R^\#_{\text{acq}}(x, v)\} &\triangleq R(x, v) \\
\{W^\#_{\text{acq}}(x, v)\} &\triangleq W(x, v) \\
\{F^\#_{\text{acq}} \} &\triangleq \{F^\#_{\text{rel}} \} \triangleq F_{\text{cross sync}} \\
\{F^\#_{\text{sync}} \} &\triangleq F_{\text{sync}} \\
\end{align*}
\]
- \( G.\text{rmw} = G_p.\text{rmw} \), \( G.\text{data} = G_p.\text{data} \), and \( G.\text{addr} = G_p.\text{addr} \)
  (the compilation does not change RMW pairs and data/address dependencies)
- \( G.\text{ctrl} \subseteq G_p.\text{ctrl} \)
  (the compilation only adds control dependencies)
- \([G.R^{\text{acq}}]; G.p \subseteq G_p.\text{rmw} \cup G_p.\text{ctrl} \)
  (a control dependency is placed from every acquire or SC read)
- \([G.R^{\text{rw}}]; G.p \subseteq G_p.\text{ctrl} \cup G_p.\text{rmw} \cap G_p.\text{data} \)
  (exclusive reads entail a control dependency to any future event, except for their immediate exclusive write successor if arose from an atomic increment)
- \( G.\text{data}; [\text{codom}(G.\text{rmw})]; G.p \subseteq G_p.\text{ctrl} \)
  (data dependency to an exclusive write entails a control dependency to any future event)
- \( G.\text{casdep}; G.p \subseteq G_p.\text{ctrl} \)
  (CAS dependency to an exclusive read entails a control dependency to any future event)

The correspondence between IMM and \( \text{POWER} \) execution graphs which follows the trailing compilation scheme may be presented similarly with two main differences. First, obviously, SC accesses are compiled to release and acquire accesses followed by SC fences:

\[
\begin{align*}
G_p.\text{E} &= G.\text{E} \cup \{n + 0.1 | n \in \{ (G.E^{\text{acq}} \setminus \text{dom}(G.\text{rmw})) \cup \text{codom}([G.R^{\text{acq}}]; G.\text{rmw}) \} \\
&\quad \cup \{n - 0.1 | n \in \{ G.W^{\text{rel}} \setminus \text{dom}(G.\text{rmw}) \} \cup \text{dom}(G.\text{rmw}; [G.W^{\text{rel}}]) \}\}
\end{align*}
\]

\[
G_p.\text{lab} = \{ e \mapsto (G.\text{lab}(e)) | e \in G.\text{E} \cup \{n + 0.1 \mapsto F^{\text{sync}} | n + 0.1 \in G_p.\text{E} \land n \in \mathbb{N} \land \\
\quad n \in G.R^{\text{acq}} \cup \text{codom}([G.R^{\text{acq}}]; G.\text{rmw}) \} \cup \{n + 0.1 \mapsto F^{\text{sync}} | n + 0.1 \in G_p.\text{E} \land n \in \mathbb{N} \land \\
\quad n \in G.E^{\text{acq}} \cup \text{codom}([G.R^{\text{acq}}]; G.\text{rmw}) \} \cup \{n - 0.1 \mapsto F^{\text{sync}} | n - 0.1 \in G_p.\text{E} \land n \in \mathbb{N}^+ \}\}
\]

Second, \([G.R^{\text{acq}}]; G.p \) has to be included in \( G_p.\text{rmw} \cup G_p.\text{ctrl} \cup G_p.p; [G_p.\text{F}^{\text{sync}}] \);
\( G_p.p^{\circ} \), not just in \( G_p.\text{rmw} \cup G_p.\text{ctrl} \), to allow for elimination of the aforementioned SC read compilation redundancy.

The next theorem ensures IMM\(_{SC}\)-consistency if the corresponding \( \text{POWER} \) execution graph is \( \text{POWER} \)-consistent.

**Theorem 5.** Let \( G \) be an IMM execution graph with whole identifiers (\( G.\text{E} \subseteq \mathbb{N} \)), and let \( G_p \) be a \( \text{POWER} \) execution graph that corresponds to \( G \). Then, \( \text{POWER}\)-consistency of \( G_p \) implies IMM\(_{SC}\)-consistency of \( G \).

**Proof (Outline).** We construct an IMM execution graph \( G' \) by inserting SC fences before SC accesses in \( G \). We also construct \( G_{\text{basics}} \) from \( G' \) by replacing SC write and read accesses of \( G' \) with release write and acquire read ones respectively.
Obviously, $\text{IMM}_\text{SC}$-consistency of $G$ follows from $\text{IMM}_\text{SC}$-consistency of $G'$, which, in turn, follows from $\text{IMM}$-consistency of $G_{\text{NoSC}}$ by Theorem 2. We construct an $\text{IMM}$ execution graph $G''$ from $G_{\text{NoSC}}$ by inserting release fences before release writes, and then an $\text{IMM}$ execution graph $G_{\text{NoRel}}$ from $G''$ by weakening the access modes of release write events to a relaxed mode. As on a previous proof step, $\text{IMM}$-consistency of $G_{\text{NoSC}}$ follows from $\text{IMM}$-consistency of $G''$, which in turn follows from $\text{IMM}$-consistency of $G_{\text{NoRel}}$ by [17, Theorem 4.1].

Thus to prove the theorem we need to show that $G_{\text{NoRel}}$ is $\text{IMM}$-consistent. Note that $G_p$—the POWER execution graph corresponding to $G$—also corresponds to $G_{\text{NoRel}}$ by construction of $G_{\text{NoRel}}$. That is, $\text{IMM}$-consistency of $G_{\text{NoRel}}$ follows from POWER-consistency of $G_p$ by [17, Theorem 4.3] since $G_{\text{NoRel}}$ does not contain SC read and write access events as well as release write access events.

$\square$