Bridging the Gap between Programming Languages and Hardware Weak Memory Models

ANTON PODKOPAEV, St. Petersburg University, JetBrains Research, Russia, and MPI-SWS, Germany
ORI LAHAV, Tel Aviv University, Israel
VIKTOR VAFEIADIS, MPI-SWS, Germany

We develop a new intermediate weak memory model, IMM, as a way of modularizing the proofs of correctness of compilation from concurrent programming languages with weak memory consistency semantics to mainstream multi-core architectures, such as POWER and ARM. We use IMM to prove the correctness of compilation from the promising semantics of Kang et al. to POWER (thereby correcting and improving their result) and ARMv7, as well as to the recently revised ARMv8 model. Our results are mechanized in Coq, and to the best of our knowledge, these are the first machine-verified compilation correctness results for models that are weaker than x86-TSO.

CCS Concepts: • **Theory of computation → Concurrency**; • **Software and its engineering → Semantics**; **Compilers**; **Correctness**;

Additional Key Words and Phrases: Weak memory consistency, IMM, promising semantics, C11 memory model

ACM Reference Format:
Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap between Programming Languages and Hardware Weak Memory Models. *Proc. ACM Program. Lang.* 3, POPL, Article 69 (January 2019), 31 pages. https://doi.org/10.1145/3290382

1 INTRODUCTION

To support platform-independent concurrent programming, languages like C/C++11 and Java9 provide several types of memory accesses and high-level fence commands. Compilers of these languages are required to map the high-level primitives to instructions of mainstream architectures: in particular, x86-TSO [Owens et al. 2009], ARMv7 and POWER [Alglave et al. 2014], and ARMv8 [Pulte et al. 2018]. In this paper, we focus on proving the correctness of such mappings. Correctness amounts to showing that for every source program P, the set of behaviors allowed by the target architecture for the mapped program $(\!|P|\!)$ (the program obtained by pointwise mapping the instructions in P) is contained in the set of behaviors allowed by the language-level model for P. Establishing such claim is a major part of a compiler correctness proof, and it is required for demonstrating the implementability of concurrency semantics.[1]

Accordingly, it has been an active research topic. In the case of C/C++11, Batty et al. [2011] established the correctness of a mapping to x86-TSO, while Batty et al. [2012] addressed the

[1]In the rest of this paper we refer to these mappings as "compilation", leaving compiler optimizations out of our scope.

Authors' addresses: Anton Podkopaev, St. Petersburg University, St. Petersburg, Russia , JetBrains Research, St. Petersburg, Russia , MPI-SWS, Germany, anton.podkopaev@jetbrains.com; Ori Lahav, Tel Aviv University, Israel, orilahav@tau.ac.il; Viktor Vafeiadis, MPI-SWS, Saarland Informatics Campus, Germany, viktor@mpi-sws.org.

2475-1421/2019/1-ART69
https://doi.org/10.1145/3290382

mapping to POWER and ARMv7. However, the correctness claims of Batty et al. [2012] were subsequently found to be incorrect [Lahav et al. 2017; Manerkar et al. 2016], as they mishandled the combination of sequentially consistent accesses with weaker accesses. Lahav et al. [2017] developed RC11, a repaired version of C/C++11, and established (by pen-and-paper proof) the correctness of the suggested compilation schemes to x86-TSO, POWER and ARMv7. Beyond (R)C11, however, there are a number of other proposed higher-level semantics, such as JMM [Manson et al. 2005], OCaml [Dolan et al. 2018], Promise [Kang et al. 2017], LLVM [Chakraborty and Vafeiadis 2017], Linux kernel memory model [Alglave et al. 2018], AE-justification [Jeffrey and Riely 2016], Bubbly [Pichon-Pharabod and Sewell 2016], and WeakestMO [Chakraborty and Vafeiadis 2019], for which only a handful of compilation correctness results have been developed.

As witnessed by a number of known incorrect claims and proofs, these correctness results may be very difficult to establish. The difficulty stems from the typical large gap between the high-level programming language concurrency features and semantics, and the architecture ones. In addition, since hardware models differ in their strength (*e.g.,* which dependencies are preserved) and the primitives they support (barriers and atomic accesses), each hardware model may require a new challenging proof.

To address this problem, we propose to modularize the compilation correctness proof to go via an intermediate model, which we call IMM (for Intermediate Memory Model). IMM contains features akin to a language-level model (such as relaxed and release/acquire accesses as well as compare-and-swap primitives), but gives them a hardware-style declarative (a.k.a. axiomatic) semantics referring to explicit syntactic dependencies.[2] IMM is very useful for structuring the compilation proofs and for enabling proof reuse: for N language semantics and M architectures, using IMM, we can reduce the number of required results from $N \times M$ to $N + M$, and moreover each of these $N + M$ proofs is typically easier than a corresponding end-to-end proof because of a smaller semantic gap between IMM and another model than between a given language-level and hardware-level model. The formal definition of IMM contains a number of subtle points as it has to be weaker than existing hardware models, and yet strong enough to support compilation from language-level models. (We discuss these points in §3.)

As summarized in Fig. 1, besides introducing IMM and proving that it is a sound abstraction over a range of hardware memory models, we prove the correctness of compilation from fragments of C11 and RC11 without non-atomic and SC accesses (denoted by (R)C11*) and from the language-level memory model of the "promising semantics" of Kang et al. [2017] to IMM.

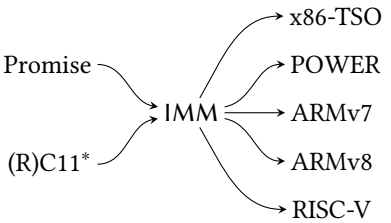


Fig. 1. Results proved in this paper.

The latter proof is the most challenging. The promising semantics is a recent prominent attempt to solve the infamous "out-of-thin-air" problem in programming language concurrency semantics [Batty et al. 2015; Boehm and Demsky 2014] without sacrificing performance. To allow efficient implementation on modern hardware platforms, the promising semantics allows threads to execute instructions out of order by having them "*promise*" (*i.e.,* pre-execute) future stores. To avoid out-of-thin-air values, every step in the promising semantics is subject to a *certification* condition. Roughly speaking, this means that thread i may take a step to a state σ, only if there exists a sequence of steps of thread i starting from σ to

[2]Being defined on a per-execution basis, IMM is not suitable as language-level semantics (see [Batty et al. 2015]). Indeed, it disallows various compiler optimizations that remove syntactic dependencies.

a state σ' in which i indeed performed (fulfilled) all its pre-executed writes (promises). Thus, the validity of a certain trace in the promising semantics depends on existence of other traces.

In mapping the promising semantics to IMM, we therefore have the largest gap to bridge: a non-standard operational semantics on the one side versus a hardware-like declarative semantics on the other side. To relate the two semantics, we carefully construct a traversal strategy on IMM execution graphs, which gives us the order in which we can execute the promising semantics machine, keep satisfying its certification condition, and finally arrive at the same outcome.

The end-to-end result is the correctness of an efficient mapping from the promising semantics of Kang et al. [2017] to the main hardware architectures. While there are two prior compilation correctness results from promising semantics to POWER and ARMv8 [Kang et al. 2017; Podkopaev et al. 2017], neither result is adequate. The POWER result [Kang et al. 2017] considered a simplified (suboptimal) compilation scheme and, in fact, we found out that its proof is *incorrect* in its handling of SC fences (see §8 for more details). In addition, its proof strategy, which is based on program transformations account for weak behaviors [Lahav and Vafeiadis 2016], cannot be applied to ARM. The ARMv8 result [Podkopaev et al. 2017] handled only a small restricted subset of the concurrency features of the promising semantics and an operational hardware model (ARMv8-POP) that was later abandoned by ARM in favor of a rather different declarative model [Pulte et al. 2018].

By encompassing all features of the promising semantics, our proof uncovered a subtle correctness problem in the conjectured compilation scheme of its read-modify-write (RMW) operations to ARMv8 and to the closely related RISC-V model. We found out that exclusive load and store operations in ARMv8 and RISC-V are weaker than those of POWER and ARMv7, following their models by Alglave et al. [2014], so that the intended compilation of RMWs is broken (see Example 3.10). Thus, the mapping to ARMv8 that we proved correct places a weak barrier (specifically ARM's "ld fence") after every RMW.[3] To keep IMM as a sound abstraction of ARMv8 and allow reuse of IMM in a future improvement of the promising semantics, we equip IMM with two types of RMWs: usual ones that are compiled to ARMv8 without the extra barrier, and stronger ones that require the extra barrier. To establish the correctness of the mapping from the (existing) promising semantics to IMM, we require that RMW instructions of the promising semantics are mapped to IMM's strong RMWs.

Finally, to ensure correctness of such subtle proofs, our results are all mechanized in Coq (~33K LOC). To the best of our knowledge, this constitutes the first mechanized correctness of compilation result from a high-level programming language concurrency model to a model weaker than x86-TSO. We believe that the existence of Coq proof scripts relating the different models may facilitate the development and investigation of weak memory models in the future, as well as the possible modifications of IMM to accommodate new and revised hardware and/or programming languages concurrency semantics.

The rest of this paper is organized as follows. In §2 we present IMM's program syntax and its mapping to execution graphs. In §3 we define IMM's consistency predicate. In §4 we present the mapping of IMM to main hardware and establish its correctness. In §5 we present the mappings from C11 and RC11 to IMM and establish their correctness. Sections 6 and 7 concern the mapping of the promising semantics of Kang et al. [2017] to IMM. To assist the reader, we discuss first (§6) a restricted fragment (with only relaxed accesses), and later (§7) extend our results and proof outline to the full promising model. Finally, we discuss related work in §8 and conclude in §9.

Supplementary material for this paper, including the Coq development, is publicly available at http://plv.mpi-sws.org/imm/.

[3]Recall that RMWs are relatively rare. The performance cost of this fixed compilation scheme is beyond the scope of this paper, and so is the improvement of the promising semantics to recover the correctness of the barrier-free compilation.

Domains

$n \in \mathbb{N}$	Natural numbers
$v \in \mathsf{Val} \triangleq \mathbb{N}$	Values
$x \in \mathsf{Loc} \triangleq \mathbb{N}$	Locations
$r \in \mathsf{Reg}$	Registers
$i \in \mathsf{Tid}$	Thread identifiers

Modes

$o_{\mathsf{R}} ::= \mathtt{rlx} \mid \mathtt{acq}$	Read modes
$o_{\mathsf{W}} ::= \mathtt{rlx} \mid \mathtt{rel}$	Write modes
$o_{\mathsf{F}} ::= \mathtt{acq} \mid \mathtt{rel} \mid \mathtt{acqrel} \mid \mathtt{sc}$	Fence modes
$o_{\mathsf{RMW}} ::= \mathtt{normal} \mid \mathtt{strong}$	RMW modes

$$\mathsf{Exp} \ni e ::= r \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid \ldots$$

$$\mathsf{Inst} \ni \mathit{inst} ::= r := e \mid \mathbf{if}\ e\ \mathbf{goto}\ n \mid [e]^{o_{\mathsf{W}}} := e \mid r := [e]^{o_{\mathsf{R}}} \mid r := \mathbf{FADD}^{o_{\mathsf{R}}, o_{\mathsf{W}}}_{o_{\mathsf{RMW}}}(e, e) \mid r := \mathbf{CAS}^{o_{\mathsf{R}}, o_{\mathsf{W}}}_{o_{\mathsf{RMW}}}(e, e, e) \mid \mathbf{fence}^{o_{\mathsf{F}}}$$

$\mathit{sprog} \in \mathsf{SProg} \triangleq \mathbb{N} \xrightarrow{\mathrm{fin}} \mathsf{Inst}$	Sequential programs
$\mathit{prog} : \mathsf{Tid} \to \mathsf{SProg}$	Programs

Fig. 2. Programming language syntax.

2 PRELIMINARIES: FROM PROGRAMS TO EXECUTION GRAPHS

Following the standard declarative (a.k.a. axiomatic) approach of defining memory consistency models [Alglave et al. 2014], the semantics of IMM programs is given in terms of *execution graphs* which partially order *events*. This is done in two steps. First, the program is mapped to a large set of execution graphs in which the read values are completely arbitrary. Then, this set is filtered by a consistency predicate, and only IMM-*consistent* execution graphs determine the possible outcomes of the program under IMM. Next, we define IMM's programming language (§2.1), define IMM's execution graphs (§2.2), and present the construction of execution graphs from programs (§2.3). The next section (§3) is devoted to present IMM's consistency predicate.

Before we start we introduce some notation for relations and functions. Given a binary relation R, we write $R^?$, R^+, and R^* respectively to denote its reflexive, transitive, and reflexive-transitive closures. The inverse relation is denoted by R^{-1}, and $\mathit{dom}(R)$ and $\mathit{codom}(R)$ denote R's domain and codomain. We denote by $R_1 ; R_2$ the left composition of two relations R_1, R_2, and assume that ; binds tighter than $\cup$ and $\setminus$. We write $R|_{\mathrm{imm}}$ for the set of all *immediate* R *edges*: $R|_{\mathrm{imm}} \triangleq R \setminus R ; R$. We denote by $[A]$ the identity relation on a set A. In particular, $[A] ; R ; [B] = R \cap (A \times B)$. For finite sets $\{a_1, \ldots, a_n\}$, we omit the set parentheses and write $[a_1, \ldots, a_n]$. Finally, for a function $f : A \to B$ and a set $X \subseteq A$, we write $f[X]$ to denote the set $\{f(x) \mid x \in X\}$.

2.1 Programming Language

IMM is formulated over the language defined in Fig. 2 with C/C++11-like concurrency features. Expressions are constructed from registers (local variables) and integers, and represent values and locations. Instructions include assignments and conditional branching, as well as memory operations. Intuitively speaking, an assignment $r := e$ assigns the value of e to register r (involving no memory access); $\mathbf{if}\ e\ \mathbf{goto}\ n$ jumps to line n of the program iff the value of e is not 0; the write $[e_1]^{o_{\mathsf{W}}} := e_2$ stores the value of e_2 in the address given by e_1; the read $r := [e]^{o_{\mathsf{R}}}$ loads the value in address e to register r; $r := \mathbf{FADD}^{o_{\mathsf{R}}, o_{\mathsf{W}}}_{o_{\mathsf{RMW}}}(e_1, e_2)$ atomically increments the value in address e_1 by the value of e_2 and loads the old value to r; $r := \mathbf{CAS}^{o_{\mathsf{R}}, o_{\mathsf{W}}}_{o_{\mathsf{RMW}}}(e, e_{\mathsf{R}}, e_{\mathsf{W}})$ atomically compares the value stored in address e to the value of e_{R}, and if the two values are the same, it replaces the value stored in e by the value of e_{W}; and fence instructions $\mathbf{fence}^{o_{\mathsf{F}}}$ are used to place global barriers.

The memory operations are annotated with *modes* that are ordered as follows:

$$\sqsubseteq \triangleq \{\langle \mathtt{rlx}, \mathtt{acq}\rangle, \langle \mathtt{rlx}, \mathtt{rel}\rangle, \langle \mathtt{acq}, \mathtt{acqrel}\rangle, \langle \mathtt{rel}, \mathtt{acqrel}\rangle, \langle \mathtt{acqrel}, \mathtt{sc}\rangle\}^*$$

Whenever $o_1 \sqsubseteq o_2$, we say that o_2 is stronger than o_1: it provides more consistency guarantees but is more costly to implement. RMWs include two modes—o_{R} for the read part and o_{W} for the write part—as well as a third (binary) mode $o_{\mathtt{RMW}}$ used to denote certain RMWs as stronger ones.

In turn, sequential programs are finite maps from $\mathbb{N}$ to instructions, and (concurrent) programs are top-level parallel composition of sequential programs, defined as mappings from a finite set Tid of thread identifiers to sequential programs. In our examples, we write sequential programs as sequences of instructions delimited by ';' (or line breaks) and use '||' for parallel composition.

Remark 1. C/C++11 sequentially consistent (SC) accesses are not included in IMM. They can be simulated, nevertheless, using SC fences following the compilation scheme of C/C++11 (see [Lahav et al. 2017]). We note that SC accesses are also not supported by the promising semantics.

2.2 Execution Graphs

Definition 2.1. An *event*, $e \in \mathsf{Event}$, takes one of the following forms:

- Non-initialization event: $\langle i, n\rangle$ where $i \in \mathsf{Tid}$ is a thread identifier, and $n \in \mathbb{Q}$ is a serial number inside each thread.
- Initialization event: $\langle \mathtt{init}\ x\rangle$ where $x \in \mathsf{Loc}$ is the location being initialized.

We denote by Init the set of all initialization events. The functions `tid` and `sn` return the (non-initialization) event's thread identifier and serial number.

Our representation of events induces a *sequenced-before* partial order on events given by:

$$e_1 < e_2 \Leftrightarrow (e_1 \in \mathsf{Init} \wedge e_2 \notin \mathsf{Init}) \vee (e_1 \notin \mathsf{Init} \wedge e_2 \notin \mathsf{Init} \wedge \mathtt{tid}(e_1) = \mathtt{tid}(e_2) \wedge \mathtt{sn}(e_1) < \mathtt{sn}(e_2))$$

Initialization events precede all non-initialization events, while events of the same thread are ordered according to their serial numbers. We use rational numbers as serial numbers to be able to easily add an event between any two events.

Definition 2.2. A *label*, $l \in \mathsf{Lab}$, takes one of the following forms:

- Read label: $\mathsf{R}_s^{o_{\mathsf{R}}}(x, v)$ where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, $o_{\mathsf{R}} \in \{\mathtt{rlx}, \mathtt{acq}\}$, and $s \in \{\mathtt{not\text{-}ex}, \mathtt{ex}\}$.
- Write label: $\mathsf{W}^{o_{\mathsf{W}}}_{o_{\mathtt{RMW}}}(x, v)$ where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, $o_{\mathsf{W}} \in \{\mathtt{rlx}, \mathtt{rel}\}$, and $o_{\mathtt{RMW}} \in \{\mathtt{normal}, \mathtt{strong}\}$.
- Fence label: $\mathsf{F}^{o_{\mathsf{F}}}$ where $o_{\mathsf{F}} \in \{\mathtt{acq}, \mathtt{rel}, \mathtt{acqrel}, \mathtt{sc}\}$.

Read labels include a location, a value, and a mode, as well as an "is exclusive" flag s. Exclusive reads stem from an RMW and are usually followed by a corresponding write. An exception is the case of a "failing" CAS (when the read value is not the expected one), where the exclusive read is not followed by a corresponding write. Write labels include a location, a value, and a mode, as well as a flag marking certain writes as strong. This will be used to differentiate the strong RMWs from the normal ones. Finally, a fence label includes just a mode.

Definition 2.3. An *execution* G consists of:

(1) a finite set $G.\mathsf{E}$ of events. Using $G.\mathsf{E}$ and the partial order $<$ on events, we derive the *program order* (a.k.a. *sequenced-before*) relation in G: $G.\mathsf{po} \triangleq [G.\mathsf{E}]; <; [G.\mathsf{E}]$. For $i \in \mathsf{Tid}$, we denote by $G.\mathsf{E}_i$ the set $\{a \in G.\mathsf{E} \mid \mathtt{tid}(a) = i\}$, and by $G.\mathsf{E}_{\neq i}$ the set $\{a \in G.\mathsf{E} \mid \mathtt{tid}(a) \neq i\}$.

(2) a labeling function $G.\mathtt{lab} : G.\mathsf{E} \to \mathsf{Lab}$. The labeling function naturally induces functions $G.\mathtt{mod}$, $G.\mathtt{loc}$, and $G.\mathtt{val}$ that return (when applicable) an event's label mode, location, and value. We use $G.\mathsf{R}, G.\mathsf{W}, G.\mathsf{F}$ to denote the subsets of $G.\mathsf{E}$ of events labeled with the respective type. We use obvious notations to further restrict the different modifiers of the event (*e.g.*, $G.\mathsf{W}(x) = \{w \in G.\mathsf{W} \mid G.\mathtt{loc}(w) = x\}$ and $G.\mathsf{F}^{\sqsupseteq o} = \{f \in G.\mathsf{F} \mid G.\mathtt{mod}(f) \sqsupseteq o\}$). We assume that $G.\mathtt{lab}(\langle \mathtt{init}\ x\rangle) = \mathsf{W}^{\mathtt{rlx}}_{\mathtt{normal}}(x, 0)$ for every $\langle \mathtt{init}\ x\rangle \in G.\mathsf{E}$.

When $sprog(pc) = \ldots$	we have the following constraints relating $pc, pc', \Phi, \Phi', G, G', \Psi, \Psi', S, S'$:
$r := e$	$pc' = pc + 1 \wedge \Phi' = \Phi[r := \Phi(e)] \wedge G' = G \wedge \Psi' = \Psi[r := \Psi(e)] \wedge S' = S$
if e **goto** n	$(\Phi(e) \neq 0 \Rightarrow pc' = n) \wedge (\Phi(e) = 0 \Rightarrow pc' = pc + 1) \wedge$ $G = G' \wedge \Phi = \Phi' \wedge \Psi' = \Psi \wedge S' = S \cup \Psi(e)$
$[e_1]^{o_W} := e_2$	$G' = \mathsf{add}_G(i, \mathsf{W}^{o_W}_{\mathsf{normal}}(\Phi(e_1), \Phi(e_2)), \emptyset, \Psi(e_2), \Psi(e_1), S, \emptyset) \wedge$ $pc' = pc + 1 \wedge \Phi' = \Phi \wedge \Psi' = \Psi \wedge S' = S$
$r := [e]^{o_R}$	$\exists v.\ G' = \mathsf{add}_G(i, \mathsf{R}^{o_R}_{\mathsf{not\text{-}ex}}(\Phi(e), v), \emptyset, \emptyset, \Psi(e), S, \emptyset) \wedge$ $pc' = pc + 1 \wedge \Phi' = \Phi[r := v] \wedge \Psi' = \Psi[r := \{\langle i, \mathsf{next}_G\rangle\}] \wedge S' = S$
$r := \mathbf{FADD}^{o_R, o_W}_{o_{RMW}}(e_1, e_2)$	$\exists v.\ \mathbf{let}\ a_R, G_R = \langle i, \mathsf{next}_G\rangle, \mathsf{add}_G(i, \mathsf{R}^{o_R}_{\mathsf{ex}}(\Phi(e_1), v), \emptyset, \emptyset, \Psi(e_1), S, \emptyset)\ \mathbf{in}$ $G' = \mathsf{add}_{G_R}(i, \mathsf{W}^{o_W}_{o_{RMW}}(\Phi(e_1), v + \Phi(e_2)), \{a_R\}, \{a_R\} \cup \Psi(e_2), \Psi(e_1), S, \emptyset) \wedge$ $pc' = pc + 1 \wedge \Phi' = \Phi[r := v] \wedge \Psi' = \Psi[r := \{a_R\}] \wedge S' = S$
$r := \mathbf{CAS}^{o_R, o_W}_{o_{RMW}}(e, e_R, e_W)$	$\exists v.\ \mathbf{let}\ a_R, G_R = \langle i, \mathsf{next}_G\rangle, \mathsf{add}_G(i, \mathsf{R}^{o_R}_{\mathsf{ex}}(\Phi(e), v), \emptyset, \emptyset, \Psi(e), S, \Psi(e_R))\ \mathbf{in}$ $pc' = pc + 1 \wedge \Phi' = \Phi[r := v] \wedge \Psi' = \Psi[r := \{a_R\}] \wedge S' = S \wedge$ $(v \neq \Phi(e_R) \Rightarrow G' = G_R) \wedge$ $(v = \Phi(e_R) \Rightarrow G' = \mathsf{add}_{G_R}(i, \mathsf{W}^{o_W}_{o_{RMW}}(\Phi(e), \Phi(e_W)), \{a_R\}, \Psi(e_W), \Psi(e), S, \emptyset))$
$\mathbf{fence}^{o_F}$	$G' = \mathsf{add}_G(i, \mathsf{F}^{o_F}, \emptyset, \emptyset, \emptyset, S, \emptyset) \wedge pc' = pc + 1 \wedge \Phi' = \Phi \wedge \Psi' = \Psi \wedge S' = S$

Fig. 3. The relation $\langle sprog, pc, \Phi, G, \Psi, S\rangle \rightarrow_i \langle sprog, pc', \Phi', G', \Psi', S'\rangle$ representing a step of thread i.

(3) a relation $G.\mathtt{rmw} \subseteq \bigcup_{x \in \mathsf{Loc}} [G.\mathsf{R}_{\mathsf{ex}}(x)]; G.\mathsf{po}|_{\mathsf{imm}}; [G.\mathsf{W}(x)]$, called *RMW pairs*. We require that $G.\mathsf{W}_{\mathtt{strong}} \subseteq \mathit{codom}(G.\mathtt{rmw})$.
(4) a relation $G.\mathtt{data} \subseteq [G.\mathsf{R}]; G.\mathsf{po}; [G.\mathsf{W}]$, called *data dependency*.
(5) a relation $G.\mathtt{addr} \subseteq [G.\mathsf{R}]; G.\mathsf{po}; [G.\mathsf{R} \cup G.\mathsf{W}]$, called *address dependency*.
(6) a relation $G.\mathtt{ctrl} \subseteq [G.\mathsf{R}]; G.\mathsf{po}$, called *control dependency*, that is forwards-closed under the program order: $G.\mathtt{ctrl}; G.\mathsf{po} \subseteq G.\mathtt{ctrl}$.
(7) a relation $G.\mathtt{casdep} \subseteq [G.\mathsf{R}]; G.\mathsf{po}; [G.\mathsf{R}_{\mathsf{ex}}]$, called *CAS dependency*.
(8) a relation $G.\mathtt{rf} \subseteq \bigcup_{x \in \mathsf{Loc}} G.\mathsf{W}(x) \times G.\mathsf{R}(x)$, called *reads-from*, and satisfying: $G.\mathtt{val}(w) = G.\mathtt{val}(r)$ for every $\langle w, r\rangle \in G.\mathtt{rf}$; and $w_1 = w_2$ whenever $\langle w_1, r\rangle, \langle w_2, r\rangle \in G.\mathtt{rf}$ (that is, $G.\mathtt{rf}^{-1}$ is functional).
(9) a strict partial order $G.\mathtt{co} \subseteq \bigcup_{x \in \mathsf{Loc}} G.\mathsf{W}(x) \times G.\mathsf{W}(x)$, called *coherence order* (a.k.a. *modification order*).

2.3 Mapping Programs to Executions

Sequential programs are mapped to execution graphs by means of an operational semantics. Its states have the form $\sigma = \langle sprog, pc, \Phi, G, \Psi, S\rangle$, where $sprog$ is the thread's sequential program; $pc \in \mathbb{N}$ points to the next instruction in $sprog$ to be executed; $\Phi : \mathsf{Reg} \to \mathsf{Val}$ maps register names to the values they store (extended to expressions in the obvious way); G is an execution graph (denoted by $\sigma.\mathsf{G}$); $\Psi : \mathsf{Reg} \to \mathcal{P}(G.\mathsf{R})$ maps each register name to the set of events that were used to compute the register's value; and $S \subseteq G.\mathsf{R}$ maintains the set of events having a control dependency to the current program point. The Ψ and S components are used to calculate the dependency edges in G. Ψ is extended to expressions in the obvious way (*e.g.*, $\Psi(n) \triangleq \emptyset$ and $\Psi(e_1 + e_2) \triangleq \Psi(e_1) \cup \Psi(e_2)$). Note that the executions graphs produced by this semantics represent traces of one thread, and as such, they are quite degenerate: $G.\mathsf{po}$ totally orders $G.\mathsf{E}$ and $G.\mathtt{rf} = G.\mathtt{co} = \emptyset$.

The *initial state* is $\sigma_0(sprog) \triangleq \langle sprog, 0, \lambda r.\ 0, G_\emptyset, \lambda r.\ \emptyset, \emptyset\rangle$ ($G_\emptyset$ denotes the empty execution), *terminal* states are those in which $pc \notin \mathit{dom}(sprog)$, and the transition relation is given in Fig. 3. It uses the notations next_G to obtain the next serial number in a thread execution graph G ($\mathsf{next}_G \triangleq |G.\mathsf{E}|$) and add_G to append an event with thread identifier i and label l to G:

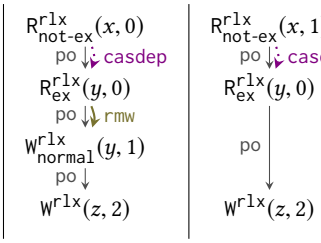
Definition 2.4. For an execution graph G, $i \in \mathsf{Tid}$, $l \in \mathsf{Lab}$, and $E_{\mathtt{rmw}}, E_{\mathtt{data}}, E_{\mathtt{addr}}, E_{\mathtt{ctrl}}, E_{\mathtt{casdep}} \subseteq G.\mathsf{R}$, $\mathsf{add}_G(i, l, E_{\mathtt{rmw}}, E_{\mathtt{data}}, E_{\mathtt{addr}}, E_{\mathtt{ctrl}}, E_{\mathtt{casdep}})$ denotes the execution graph G' given by:

$$G'.\mathsf{E} = G.\mathsf{E} \uplus \{\langle i, \mathsf{next}_G\rangle\} \qquad G'.\mathtt{lab} = G.\mathtt{lab} \uplus \{\langle i, \mathsf{next}_G\rangle \mapsto l\}$$
$$G'.\mathtt{rmw} = G.\mathtt{rmw} \uplus (E_{\mathtt{rmw}} \times \{\langle i, \mathsf{next}_G\rangle\}) \qquad G'.\mathtt{data} = G.\mathtt{data} \uplus (E_{\mathtt{data}} \times \{\langle i, \mathsf{next}_G\rangle\})$$
$$G'.\mathtt{addr} = G.\mathtt{addr} \uplus (E_{\mathtt{addr}} \times \{\langle i, \mathsf{next}_G\rangle\}) \qquad G'.\mathtt{ctrl} = G.\mathtt{ctrl} \uplus (E_{\mathtt{ctrl}} \times \{\langle i, \mathsf{next}_G\rangle\})$$
$$G'.\mathtt{casdep} = G.\mathtt{casdep} \uplus (E_{\mathtt{casdep}} \times \{\langle i, \mathsf{next}_G\rangle\}) \qquad G'.\mathtt{rf} = G.\mathtt{rf} \qquad G'.\mathtt{co} = G.\mathtt{co}$$

Besides the explicit calculation of dependencies, the operational semantics is standard.

Example 2.5. The only novel ingredient is the *CAS dependency* relation, which tracks reads that affect the success of a CAS instruction. As an example, consider the following program.

$$a := [x]^{\mathtt{rlx}}$$
$$b := \mathbf{CAS}^{\mathtt{rlx},\mathtt{rlx}}_{\mathtt{normal}}(y, a, 1)$$
$$[z]^{\mathtt{rlx}} := 2$$



The CAS instruction may produce a write event or not, depending on the value read from y and the value of register a, which is assigned at the read instruction from x. The casdep edge reflects the latter dependency in both representative execution graphs. The mapping of IMM's CAS instructions to POWER and ARM ensures that the casdep on the source execution graph implies a control dependency to all po-later events in the target graph (see §4). □

Next, we define *program executions*.

Definition 2.6. For an execution graph G and $i \in \mathsf{Tid}$, $G|_i$ denotes the execution graph given by:

$$G|_i.\mathsf{E} = G.\mathsf{E}_i \qquad G|_i.\mathtt{lab} = G.\mathtt{lab}|_{G.\mathsf{E}_i}$$
$$G|_i.\mathtt{rmw} = [G.\mathsf{E}_i]; G.\mathtt{rmw}; [G.\mathsf{E}_i] \qquad G|_i.\mathtt{data} = [G.\mathsf{E}_i]; G.\mathtt{data}; [G.\mathsf{E}_i]$$
$$G|_i.\mathtt{addr} = [G.\mathsf{E}_i]; G.\mathtt{addr}; [G.\mathsf{E}_i] \qquad G|_i.\mathtt{ctrl} = [G.\mathsf{E}_i]; G.\mathtt{ctrl}; [G.\mathsf{E}_i]$$
$$G|_i.\mathtt{casdep} = [G.\mathsf{E}_i]; G.\mathtt{casdep}; [G.\mathsf{E}_i] \qquad G|_i.\mathtt{rf} = G|_i.\mathtt{co} = \emptyset$$

Definition 2.7 (Program executions). An execution graph G is a *(full) execution graph of a program prog* if for every $i \in \mathsf{Tid}$, there exists a (terminal) state σ such that $\sigma.\mathsf{G} = G|_i$ and $\sigma_0(prog(i)) \rightarrow_i^* \sigma$.

Now, given the IMM-consistency predicate presented in the next section, we define the set of allowed outcomes.

Definition 2.8. G is *initialized* if $\langle \mathtt{init}\ x\rangle \in G.\mathsf{E}$ for every $x \in G.\mathtt{loc}[G.\mathsf{E}]$.

Definition 2.9. A function $O : \mathsf{Loc} \to \mathsf{Val}$ is:

- an *outcome of an execution graph* G if for every $x \in \mathsf{Loc}$, either $O(x) = G.\mathtt{val}(w)$ for some $G.\mathtt{co}$-maximal event $w \in G.\mathsf{W}(x)$, or $O(x) = 0$ and $G.\mathsf{W}(x) = \emptyset$.
- an *outcome of a program prog* under IMM if O is an outcome of some IMM-consistent initialized full execution graph of *prog*.

3 IMM: THE INTERMEDIATE MODEL

In this section, we introduce the consistency predicate of IMM. The first (standard) conditions require that every read reads from some write ($codom(G.\mathtt{rf}) = G.\mathsf{R}$), and that the coherence order totally orders the writes to each location ($G.\mathtt{co}$ totally orders $G.\mathsf{W}(x)$ for every $x \in \mathsf{Loc}$). In addition, we require (1) coherence, (2) atomicity of RMWs, and (3) global ordering, which are formulated in the rest of this section, with the help of several derived relations on events.

The rest of this section is described in the context of a given execution graph G, and the 'G.' prefix is omitted. In addition, we employ the following notational conventions: for every relation $\mathtt{x} \subseteq \mathtt{E} \times \mathtt{E}$, we denote by $\mathtt{xe}$ its thread external restriction ($\mathtt{xe} \triangleq \mathtt{x} \setminus \mathtt{po}$), while $\mathtt{xi}$ denotes its thread internal restriction ($\mathtt{xi} \triangleq \mathtt{x} \cap \mathtt{po}$). We denote by $\mathtt{x}|_{\mathtt{loc}}$ its restriction to accesses to the same location ($\mathtt{x}|_{\mathtt{loc}} \triangleq \bigcup_{x \in \mathtt{Loc}} [\mathtt{R}(x) \cup \mathtt{W}(x)] ; \mathtt{x} ; [\mathtt{R}(x) \cup \mathtt{W}(x)]$).

3.1 Coherence

Coherence is a basic property of memory models that implies that programs with only one shared location behave as if they were running under sequential consistency. Hardware memory models typically enforce coherence by requiring that $\mathtt{po}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{rf}^{-1} ; \mathtt{co}$ is acyclic (a.k.a. *SC-per-location*). Language models, however, strengthen the coherence requirement by replacing po with a "happens before" relation hb that includes po as well as inter-thread synchronization. Since IMM's purpose is to verify the implementability of language-level models, we take its coherence axiom to be close to those of language-level models. Following [Lahav et al. 2017], we therefore define the following relations:

$$\mathtt{rs} \triangleq [\mathtt{W}] ; \mathtt{po}|_{\mathtt{loc}} ; [\mathtt{W}] \cup [\mathtt{W}] ; (\mathtt{po}|_{\mathtt{loc}}^{?} ; \mathtt{rf} ; \mathtt{rmw})^{*} \qquad \textit{(release sequence)}$$

$$\mathtt{release} \triangleq ([\mathtt{W}^{\mathtt{rel}}] \cup [\mathtt{F}^{\sqsupseteq \mathtt{rel}}] ; \mathtt{po}) ; \mathtt{rs} \qquad \textit{(release prefix)}$$

$$\mathtt{sw} \triangleq \mathtt{release} ; (\mathtt{rfi} \cup \mathtt{po}|_{\mathtt{loc}}^{?} ; \mathtt{rfe}) ; ([\mathtt{R}^{\mathtt{acq}}] \cup \mathtt{po} ; [\mathtt{F}^{\sqsupseteq \mathtt{acq}}]) \qquad \textit{(synchronizes with)}$$

$$\mathtt{hb} \triangleq (\mathtt{po} \cup \mathtt{sw})^{+} \qquad \textit{(happens-before)}$$

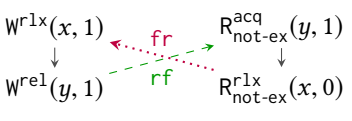
$$\mathtt{fr} \triangleq \mathtt{rf}^{-1} ; \mathtt{co} \qquad \textit{(from-read/read-before)}$$

$$\mathtt{eco} \triangleq \mathtt{rf} \cup \mathtt{co} ; \mathtt{rf}^{?} \cup \mathtt{fr} ; \mathtt{rf}^{?} \qquad \textit{(extended coherence order)}$$

We say that G is coherent if $\mathtt{hb} ; \mathtt{eco}^{?}$ is irreflexive, or equivalently $\mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr}$ is acyclic.

Example 3.1 (Message passing). Coherence disallows the weak behavior of the MP litmus test:

$$\begin{array}{l|l} [x]^{\mathtt{rlx}} := 1 & a := [y]^{\mathtt{acq}} \;\; /\!/ 1 \\ [y]^{\mathtt{rel}} := 1 & b := [x]^{\mathtt{rlx}} \;\; /\!/ 0 \end{array}$$



To the right, we present the execution yielding the annotated weak outcome.[4] The rf-edges and the induced fr-edge are determined by the annotated outcome. The displayed execution is inconsistent because the rf-edge between the release write and the acquire read constitutes an sw-edge, and hence there is an $\mathtt{hb} ; \mathtt{fr}$ cycle. □

Remark 2. Adept readers may notice that our definition of sw is stronger (namely, our sw is larger) than the one of RC11 [Lahav et al. 2017], which (following the fixes of Vafeiadis et al. [2015] to C/C++11's original definition) employs the following definitions:

$$\mathtt{rs}_{\mathrm{RC11}} \triangleq [\mathtt{W}] ; \mathtt{po}|_{\mathtt{loc}}^{?} ; (\mathtt{rf} ; \mathtt{rmw})^{*} \qquad \mathtt{release}_{\mathrm{RC11}} \triangleq ([\mathtt{W}^{\mathtt{rel}}] \cup [\mathtt{F}^{\sqsupseteq \mathtt{rel}}] ; \mathtt{po}) ; \mathtt{rs}_{\mathrm{RC11}}$$

$$\mathtt{sw}_{\mathrm{RC11}} \triangleq \mathtt{release} ; \mathtt{rf} ; ([\mathtt{R}^{\mathtt{acq}}] \cup \mathtt{po} ; [\mathtt{F}^{\sqsupseteq \mathtt{acq}}]) \qquad \mathtt{hb}_{\mathrm{RC11}} \triangleq (\mathtt{po} \cup \mathtt{sw}_{\mathrm{RC11}})^{+}$$

The reason for this discrepancy is our aim to allow the splitting of release writes and RMWs into release fences followed by relaxed operations. Indeed, as explained in §4.1, the soundness

[4] We use program comments notation to refer to the read values in the behavior we discuss. These can be formally expressed as program outcomes (Def. 2.9) by storing the read values in distinguished memory locations. In addition, for conciseness, we do not show the implicit initialization events and the rf and co edges from them, and include the $o_{\mathtt{RMW}}$ subscript only for writes in $\mathit{codom}(G.\mathtt{rmw})$ (recall that $G.\mathtt{W}_{\mathtt{strong}} \subseteq \mathit{codom}(G.\mathtt{rmw})$).

of this transformation allows us to simplify our proofs. In RC11 [Lahav et al. 2017], as well as in C/C++11 [Batty et al. 2011], this rather intuitive transformation, as we found out, is actually *unsound*. To see this consider the following example:

$$\begin{array}{l|l|l} [y]^{\mathtt{rlx}} := 1 & a := \mathbf{FADD}^{\mathtt{acq,rel}}(x,1) \;/\!/\, 1 & b := [x]^{\mathtt{acq}} \;/\!/\, 3 \\ [x]^{\mathtt{rel}} := 1 & [x]^{\mathtt{rlx}} := 3 & c := [y]^{\mathtt{rlx}} \;/\!/\, 0 \end{array}$$

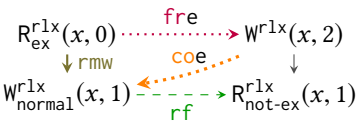
(R)C11 disallows the annotated behavior, due in particular to the release sequence formed from the release exclusive write to x in the second thread to its subsequent relaxed write. However, if we split the increment to $\mathbf{fence}^{\mathtt{rel}}; a := \mathbf{FADD}^{\mathtt{acq,rlx}}(x,1)$ (which intuitively may seem stronger), the release sequence will no longer exist, and the annotated behavior will be allowed. IMM overcomes this problem by strengthening sw in a way that ensures a synchronization edge for the transformed program as well. In §4.1, we establish the soundness of this splitting transformation in general. In addition, note that, as we show in §4, existing hardware support IMM's stronger synchronization without strengthening the intended compilation schemes. On the other hand, in our proof concerning the promising semantics in §7, it is more convenient to use RC11's definition of sw, which results in a (provably) stronger (namely, allowing less behaviors) model that still accounts for all the behaviors of the promising semantics.[5]

3.2 RMW Atomicity

Atomicity of RMWs simply states that the load of a successful RMW reads from the immediate co-preceding write before the RMW's store. Formally, $\mathtt{rmw} \cap (\mathtt{fre} ; \mathtt{coe}) = \emptyset$, which says that there is no other write ordered between the load and the store of an RMW.

Example 3.2 (Violation of RMW atomicity). The following behavior violates the fetch-and-add atomicity and is disallowed by all known weak memory models.

$$a := \mathbf{FADD}^{\mathtt{rlx,rlx}}_{\mathtt{normal}}(x,1) \;/\!/\, 0 \;\Big\|\; [x]^{\mathtt{rlx}} := 2 \;\Big\|\; b := [x]^{\mathtt{rlx}} \;/\!/\, 1$$



To the right, we present an inconsistent execution corresponding to the outcome omitting the initialization event for conciseness. The rf edges and the induced fre edge are forced by the annotated outcome, while the coe edge is forced because of coherence: *i.e.,* ordering the writes in the reverse order yields a coherence violation. The atomicity violation is thus evident. □

3.3 Global Ordering Constraint

The third condition—the *global ordering* constraint—is the most complicated and is used to rule out out-of-thin-air behaviors. We will incrementally define a relation ar that we require to be acyclic.

First of all, ar includes the external reads-from relation, rfe, and the ordering guarantees induced by memory fences and release/acquire accesses. Specifically, release writes enforce an ordering to any previous event of the same thread, acquire reads enforce the ordering to subsequent events of the same thread, while fences are ordered with respect to both prior and subsequent events. As a final condition, release writes are ordered before any subsequent writes to the same location: this is needed for maintaining release sequences.

$$\mathsf{bob} \triangleq \mathsf{po} ; [\mathsf{W}^{\mathtt{rel}}] \cup [\mathsf{R}^{\mathtt{acq}}] ; \mathsf{po} \cup \mathsf{po} ; [\mathsf{F}] \cup [\mathsf{F}] ; \mathsf{po} \cup [\mathsf{W}^{\mathtt{rel}}] ; \mathsf{po}|_{\mathtt{loc}} ; [\mathsf{W}] \qquad (\textit{barrier order})$$

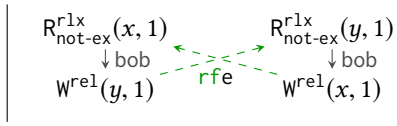
[5]The C++ committee is currently revising the release sequence definition aiming to simplify it and relate it to its actual uses. The analysis here may provide further input to that discussion.

$$\mathtt{ar} \triangleq \mathtt{rfe} \cup \mathtt{bob} \cup \ldots \qquad \text{(\textit{acyclicity relation, more cases to be added})}$$

Release/acquire accesses and fences in IMM play a double role: they induce synchronization similar to RC11 as discussed in §3.1 and also enforce intra-thread instruction ordering as in hardware models. The latter role ensures the absence of 'load buffering' behaviors in the following examples.

Example 3.3 (Load buffering with release writes). Consider the following program, whose annotated outcome disallowed by ARM, POWER, and the promising semantics.[6]

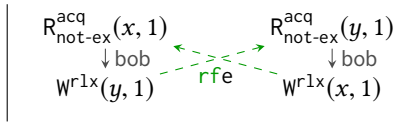
$$\begin{array}{l|l} a := [x]^{\mathtt{rlx}} \;\#1 & b := [y]^{\mathtt{rlx}} \;\#1 \\ [y]^{\mathtt{rel}} := 1 & [x]^{\mathtt{rel}} := 1 \end{array}$$



IMM disallows the outcome because of the $\mathtt{bob} \cup \mathtt{rfe}$ cycle. □

Example 3.4 (Load buffering with acquire reads). Consider a variant of the previous program with acquire loads and relaxed stores:

$$\begin{array}{l|l} a := [x]^{\mathtt{acq}} \;\#1 & b := [y]^{\mathtt{acq}} \;\#1 \\ [y]^{\mathtt{rlx}} := 1 & [x]^{\mathtt{rlx}} := 1 \end{array}$$



IMM again declares the presented execution as inconsistent following both ARM and POWER, which forbid the annotated outcome. The promising semantics, in contrast, allows this outcome to support a higher-level optimization (namely, elimination of redundant acquire reads). □

Besides orderings due to fences, hardware preserves certain orderings due to syntactic code dependencies. Specifically, whenever a write depends on some earlier read by a chain of syntactic dependencies or internal reads-from edges (which are essentially dependencies through memory), then the hardware cannot execute the write until it has finished executing the read, and so the ordering between them is preserved. We call such preserved dependency sequences the *preserved program order* (ppo) and include it in ar. In contrast, dependencies between read events are not always preserved, and so we do not incorporate them in the ar relation.

$$\mathtt{deps} \triangleq \mathtt{data} \cup \mathtt{ctrl} \cup \mathtt{addr} ; \mathtt{po}^{?} \cup \mathtt{casdep} \cup [\mathsf{R}_{\mathtt{ex}}] ; \mathtt{po} \qquad \text{(\textit{syntactic dependencies})}$$

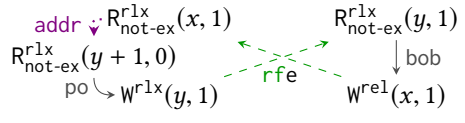
$$\mathtt{ppo} \triangleq [\mathsf{R}] ; (\mathtt{deps} \cup \mathtt{rfi})^{+} ; [\mathsf{W}] \qquad \text{(\textit{preserved program order})}$$

$$\mathtt{ar} \triangleq \mathtt{rfe} \cup \mathtt{bob} \cup \mathtt{ppo} \cup \ldots$$

The extended constraint rules out the weak behaviors of variants of the load buffering example that use syntactic dependencies to enforce an ordering.

Example 3.5 (Load buffering with an address dependency). Consider a variant of the previous program with an address-dependent read instruction in the middle of the first thread:

$$\begin{array}{l|l} a := [x]^{\mathtt{rlx}} \;\#1 & c := [y]^{\mathtt{rlx}} \;\#1 \\ b := [y + a]^{\mathtt{rlx}} & [x]^{\mathtt{rel}} := 1 \\ [y]^{\mathtt{rlx}} := 1 & \end{array}$$



The displayed execution is IMM-inconsistent because of the addr;po;rfe;bob;rfe cycle. Hardware implementations cannot produce the annotated behavior because the write to y cannot be issued

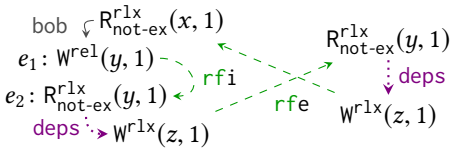
[6] In this and other examples, when saying whether a behavior of a program is allowed by ARM/POWER, we implicitly mean the intended mapping of the program's primitive accesses to ARM/POWER. See §4 for details.

until it has been determined that its address does not alias with $y + a$, which cannot be determined until the value of x has been read. □

Similar to syntactic dependencies, rfi edges are guaranteed to be preserved only on dependency paths from a read to a write, not otherwise.

Example 3.6 (rfi is not always preserved). Consider the following program, whose annotated outcome is allowed by ARMv8.

$$\begin{array}{l} a := [x]^{\mathtt{rlx}} \;\; /\!/ 1 \\ e_1 : [y]^{\mathtt{rel}} := 1 \\ e_2 : b := [y]^{\mathtt{rlx}} \;\; /\!/ 1 \\ \quad [z]^{\mathtt{rlx}} := b \end{array} \Bigg\| \begin{array}{l} c := [z]^{\mathtt{rlx}} \;\; /\!/ 1 \\ [x]^{\mathtt{rlx}} := c \end{array}$$



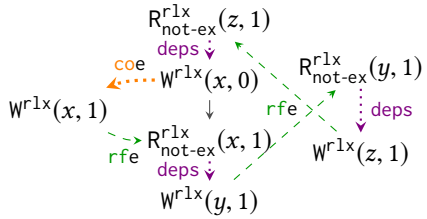
To the right, we show the corresponding execution (the rf edges are forced because of the outcome). Had we included rfi unconditionally as part of ar, we would have disallowed the behavior, because it would have introduced an ar edge between events e_1 and e_2, and therefore an ar cycle. □

Note that we do not include fri in ppo since it is not preserved in ARMv7 [Alglave et al. 2014] (unlike in x86-TSO, POWER, and ARMv8). Thus, as ARMv7 (as well as the Flowing and POP models of ARM in [Flur et al. 2016]), IMM allows the weak behavior from [Lahav and Vafeiadis 2016, §6].

Next, we include $\mathtt{detour} \triangleq (\mathtt{coe} \,;\, \mathtt{rfe}) \cap \mathtt{po}$ in ar. It captures the case when a read r does not read from an earlier write w to the same location but from a write w' of a different thread. In this case, both ARM and POWER enforce an ordering between w and r. Since the promising semantics also enforces such orderings (due to the certification requirement in every future memory, see §7), IMM also enforces the ordering by including detour in ar.

Example 3.7 (Enforcing detour). The annotated behavior of the following program is disallowed by POWER, ARM, and the promising semantics, and so it must be disallowed by IMM.

$$[x]^{\mathtt{rlx}} := 1 \Bigg\| \begin{array}{l} a := [z]^{\mathtt{rlx}} \;\; /\!/ 1 \\ [x]^{\mathtt{rlx}} := a - 1 \\ b := [x]^{\mathtt{rlx}} \;\; /\!/ 1 \\ [y]^{\mathtt{rlx}} := b \end{array} \Bigg\| \begin{array}{l} c := [y]^{\mathtt{rlx}} \;\; /\!/ 1 \\ [z]^{\mathtt{rlx}} := c \end{array}$$



If we were to exclude detour from the acyclicity condition, the execution of the program shown above to the right would have been allowed by IMM. □

We move on to a constraint about SC fences. Besides constraining the ordering of events from the same thread, SC fences induce inter-thread orderings whenever there is a coherence path between them. Following the RC11 model [Lahav et al. 2017], we call this relation psc and include it in ar.

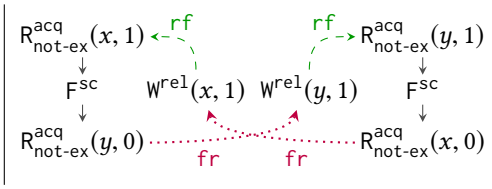
$$\mathtt{psc} \triangleq [\mathsf{F}^{\mathtt{sc}}] \,;\, \mathtt{hb} \,;\, \mathtt{eco} \,;\, \mathtt{hb} \,;\, [\mathsf{F}^{\mathtt{sc}}] \qquad \textit{(partial SC fence order)}$$

$$\mathtt{ar} \triangleq \mathtt{rfe} \cup \mathtt{bob} \cup \mathtt{ppo} \cup \mathtt{detour} \cup \mathtt{psc} \cup \ldots$$

Example 3.8 (Independent reads of independent writes). Similar to POWER, IMM is not "multi-copy atomic" [Maranget et al. 2012] (or "memory atomic" [Zhang et al. 2018]). In particular, it allows

the weak behavior of the IRIW litmus test even with release-acquire accesses. To forbid the weak behavior, one has to use SC fences:

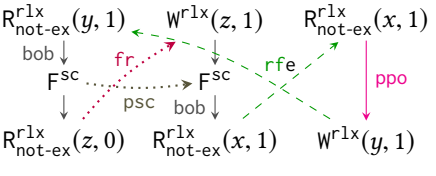
$$\begin{array}{l} a := [x]^{\mathsf{acq}} \;/\!/\, 1 \\ \mathbf{fence}^{\mathsf{sc}} \\ b := [y]^{\mathsf{acq}} \;/\!/\, 0 \end{array} \Bigg\| [x]^{\mathsf{rel}} := 1 \Bigg\| [y]^{\mathsf{rel}} := 1 \Bigg\| \begin{array}{l} c := [y]^{\mathsf{acq}} \;/\!/\, 1 \\ \mathbf{fence}^{\mathsf{sc}} \\ d := [x]^{\mathsf{acq}} \;/\!/\, 0 \end{array}$$



The execution corresponding to the weak outcome is shown to the right. For soundness w.r.t. the promising semantics, IMM declares this execution to be inconsistent (which is also natural since it has an SC fence between every two instructions). It does so due to the psc cycle: each fence reaches the other by a po ; fr ; rf ; po ⊆ psc path. When the SC fences are omitted, since POWER allows the weak outcome, IMM allows it as well. □

Example 3.9. To illustrate why we make psc part of ar, rather than a separate acyclicity condition (as in RC11), consider the following program, whose annotated outcome is forbidden by the promising semantics.

$$\begin{array}{l} a := [y]^{\mathsf{rlx}} \;/\!/\, 1 \\ \mathbf{fence}^{\mathsf{sc}} \\ b := [z]^{\mathsf{rlx}} \;/\!/\, 0 \end{array} \Bigg\| \begin{array}{l} [z]^{\mathsf{rlx}} := 1 \\ \mathbf{fence}^{\mathsf{sc}} \\ c := [x]^{\mathsf{rlx}} \;/\!/\, 1 \end{array} \Bigg\| \begin{array}{l} d := [x]^{\mathsf{rlx}} \;/\!/\, 1 \\ \mathbf{if}\ d \neq 0\ \mathbf{goto}\ L \\ [y]^{\mathsf{rlx}} := 1 \\ L\colon \end{array}$$



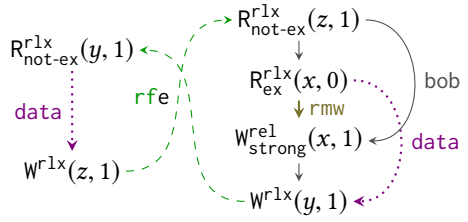
The execution corresponding to that outcome is shown to the right. For soundness w.r.t. the promising semantics, IMM declares this execution inconsistent, due to the ar cycle. □

The final case we add to ar is to support the questionable semantics of RMWs in the promising semantics. The promising semantics requires the ordering between the store of a release RMW and subsequent stores to be preserved, something that is not generally guaranteed by ARMv8. For this reason, to be able to compile the promising semantics to IMM, and still keep IMM as a sound abstraction of ARMv8, we include the additional "RMW mode" in RMW instructions, which propagates to their induced write events. Then, we include $[\mathsf{W}_{\mathsf{strong}}]$; po ; [W] in ar, yielding the following (final) definition:

$$\mathsf{ar} \triangleq \mathsf{rfe} \cup \mathsf{bob} \cup \mathsf{ppo} \cup \mathsf{detour} \cup \mathsf{psc} \cup [\mathsf{W}_{\mathsf{strong}}] ; \mathsf{po} ; [\mathsf{W}]$$

Example 3.10. The following example demonstrates the problem in the intended mapping of the promising semantics to ARMv8.

$$\begin{array}{l} a := [y]^{\mathsf{rlx}} \;/\!/\, 1 \\ [z]^{\mathsf{rlx}} := a \end{array} \Bigg\| \begin{array}{l} b := [z]^{\mathsf{rlx}} \;/\!/\, 1 \\ c := \mathbf{FADD}^{\mathsf{rlx,rel}}_{\mathsf{strong}}(x,1) \;/\!/\, 0 \\ [y]^{\mathsf{rlx}} := c + 1 \end{array}$$



The promising semantics disallows the annotated behavior (it requires a promise of $y = 1$, but this promise cannot be certified for a future memory that will not allow the atomic increment from 0—see §7.1 and Example 7.6). It is disallowed by IMM due to the ar cycle (from the read of y):

$$
\begin{array}{rl}
(\!|r := [e]^{\mathsf{rlx}}|\!) \approx \text{“}\texttt{ld}\text{”} & (\!|[e_1]^{\mathsf{rlx}} := e_2|\!) \approx \text{“}\texttt{st}\text{”} \\
(\!|r := [e]^{\mathsf{acq}}|\!) \approx \text{“}\texttt{ld;cmp;bc;isync}\text{”} & (\!|[e_1]^{\mathsf{rel}} := e_2|\!) \approx \text{“}\texttt{lwsync;st}\text{”} \\
(\!|\mathbf{fence}^{\neq \mathsf{sc}}|\!) \approx \text{“}\texttt{lwsync}\text{”} & (\!|\mathbf{fence}^{\mathsf{sc}}|\!) \approx \text{“}\texttt{sync}\text{”} \\
\multicolumn{2}{c}{(\!|r := \mathbf{FADD}^{o_{\mathsf{R}},o_{\mathsf{W}}}_{o_{\mathsf{RMW}}}(e_1,e_2)|\!) \approx \mathrm{wmod}(o_{\mathsf{W}}) +\!\!+ \text{“}\texttt{L:lwarx;stwcx.;bc L}\text{”} +\!\!+ \mathrm{rmod}(o_{\mathsf{R}})} \\
\multicolumn{2}{c}{(\!|r := \mathbf{CAS}^{o_{\mathsf{R}},o_{\mathsf{W}}}_{o_{\mathsf{RMW}}}(e,e_{\mathsf{R}},e_{\mathsf{W}})|\!) \approx \mathrm{wmod}(o_{\mathsf{W}}) +\!\!+ \text{“}\texttt{L:lwarx;cmp;bc Le;stwcx.;bc L;Le:}\text{”} +\!\!+ \mathrm{rmod}(o_{\mathsf{R}})} \\
\multicolumn{2}{c}{\mathrm{wmod}(o_{\mathsf{W}}) \triangleq o_{\mathsf{W}} = \mathtt{rel}\ ?\ \text{“}\texttt{lwsync;}\text{”}\ :\ \text{“”} \qquad \mathrm{rmod}(o_{\mathsf{R}}) \triangleq o_{\mathsf{R}} = \mathsf{acq}\ ?\ \text{“}\texttt{;isync}\text{”}\ :\ \text{“”}}
\end{array}
$$

Fig. 4. Compilation scheme from IMM to POWER.

ppo ; rfe ; bob ; $[\mathsf{W}_{\mathtt{strong}}]$; po ; [W] ; rfe. Without additional barriers, ARMv8 allows this behavior. Thus, our mapping of IMM to ARMv8 places a barrier (“ld fence”) after strong RMWs (see §4.2). □

3.4 Consistency

Putting everything together, IMM-consistency is defined as follows.

Definition 3.11. G is called IMM-*consistent* if the following hold:

- $\mathit{codom}(G.\mathtt{rf}) = G.\mathsf{R}$. (rf-completeness)
- For every location $x \in \mathsf{Loc}$, $G.\mathtt{co}$ totally orders $G.\mathsf{W}(x)$. (co-totality)
- $G.\mathtt{hb}$; $G.\mathtt{eco}^?$ is irreflexive. (coherence)
- $G.\mathtt{rmw} \cap (G.\mathtt{fre}\ ;\ G.\mathtt{coe}) = \emptyset$. (atomicity)
- $G.\mathtt{ar}$ is acyclic. (no-thin-air)

4 FROM IMM TO HARDWARE MODELS

In this section, we provide mappings from IMM to the main hardware architectures and establish their soundness. That is, if some behavior is allowed by a target architecture on a target program, then it is also allowed by IMM on the source of that program. Since the models of hardware we consider are declarative, we formulate the soundness results on the level of execution graphs, keeping the connection to programs only implicit. Indeed, a mapping of IMM instructions to real architecture instructions naturally induces a mapping of IMM execution graphs to target architecture execution graphs. Then, it suffices to establish that the consistency of a target execution graph (as defined by the target memory model) entails the IMM-consistency of its source execution graph. This is a common approach for studying declarative models, (see, *e.g.*, [Vafeiadis et al. 2015]), and allows us to avoid orthogonal details of the target architectures' instruction sets.

Next, we study the mapping to POWER (§4.1) and ARMv8 (§4.2). We note that IMM can be straightforwardly shown to be weaker than x86-TSO, and thus the identity mapping (up to different syntax) is a correct compilation scheme from IMM to x86-TSO. The mapping to ARMv7 is closely related to POWER, and it is discussed in §4.1 as well. RISC-V [RISC-V 2018; RISCV in herd 2018] is stronger than ARMv8 and therefore soundness of mapping to it from IMM follows from the corresponding ARMv8 result.

4.1 From IMM to POWER

The intended mapping of IMM to POWER is presented schematically in Fig. 4. It follows the C/C++11 mapping [Mapping 2016] (see also [Maranget et al. 2012]): relaxed reads and writes are compiled down to plain machine loads and stores; acquire reads are mapped to plain loads followed by a control dependent instruction fence; release writes are mapped to plain writes preceded by a lightweight fence; acquire/release/acquire-release fences are mapped to POWER's lightweight fences; and SC fences are mapped to full fences. The compilation of RMWs requires a loop which repeatedly uses POWER's load-reserve/store-conditional instructions until the store-conditional

succeeds. RMWs are accompanied with barriers for acquire/release modes as reads and writes. CAS instructions proceed to the conditional write only after checking that the loaded value meets the required condition. Note that IMM's strong RMWs are compiled to POWER as normal RMWs.

To simplify our correctness proof, we take advantage of the fact that release writes and release RMWs are compiled down as their relaxed counterparts with a preceding $\mathbf{fence}^{\mathtt{rel}}$. Thus, we consider the compilation as if it happens in two steps: first, release writes and RMWs are split to release fences and their relaxed counterparts; and then, the mapping of Fig. 4 is applied (for a program without release writes and release RMWs). Accordingly, we establish (*i*) the soundness of the split of release accesses; and (*ii*) the correctness of the mapping in the absence of release accesses.[7] The first obligation is solely on the side of IMM, and is formally presented next.

Theorem 4.1. *Let G be an* IMM *execution graph such that* $G.\mathtt{po} ; [G.\mathtt{W}^{\mathtt{rel}}] \subseteq G.\mathtt{po}^{?} ; [G.\mathtt{F}^{\mathtt{rel}}] ; G.\mathtt{po} \cup G.\mathtt{rmw}$. *Let G' be the* IMM *execution graph obtained from G by weakening the access modes of release write events to a relaxed mode. Then,* IMM*-consistency of G' implies* IMM*-consistency of G.*

Next, we establish the correctness of the mapping (in the absence of release writes) with respect to the model of the POWER architecture of Alglave et al. [2014], which we denote by POWER. As IMM, the POWER model is declarative, defining allowed outcomes via consistent execution graphs. Its labels are similar to IMM's labels (Def. 2.2) with the following exceptions:

- Read/write labels have the form $\mathtt{R}(x, v)$ and $\mathtt{W}(x, v)$: they do not include additional modes.
- There are three fence labels (listed here in increasing strength order): an "instruction fence" ($\mathtt{F}^{\mathtt{isync}}$), a "lightweight fence" ($\mathtt{F}^{\mathtt{lwsync}}$), and a "full fence" ($\mathtt{F}^{\mathtt{sync}}$).

In turn, POWER execution graphs are defined as those of IMM (cf. Def. 2.3), except for the CAS dependency, casdep, which is not present in POWER executions. The next definition presents the correspondence between IMM execution graphs and their mapped POWER ones following the compilation scheme in Fig. 4.

Definition 4.2. Let G be an IMM execution graph with whole serial numbers ($\mathsf{sn}[G.\mathtt{E}] \subseteq \mathbb{N}$), such that $G.\mathtt{W}^{\mathtt{rel}} = \emptyset$. A POWER execution graph G_p *corresponds* to G if the following hold:

- $G_p.\mathtt{E} = G.\mathtt{E} \cup \{\langle i, n + 0.5\rangle \mid \langle i, n\rangle \in (G.\mathtt{R}^{\mathtt{acq}} \setminus \mathit{dom}(G.\mathtt{rmw})) \cup \mathit{codom}([G.\mathtt{R}^{\mathtt{acq}}] ; G.\mathtt{rmw})\}$
  (new events are added after acquire reads and acquire RMW pairs)
- $G_p.\mathtt{lab} = \{e \mapsto (\!|G.\mathtt{lab}(e)|\!) \mid e \in G.\mathtt{E}\} \cup \{e \mapsto \mathtt{F}^{\mathtt{isync}} \mid e \in G_p.\mathtt{E} \setminus G.\mathtt{E}\}$ where:

$$(\!|\mathtt{R}^{o_{\mathtt{R}}}_{s}(x, v)|\!) \triangleq \mathtt{R}(x, v) \qquad (\!|\mathtt{F}^{\mathtt{acq}}|\!) = (\!|\mathtt{F}^{\mathtt{rel}}|\!) = (\!|\mathtt{F}^{\mathtt{acqrel}}|\!) \triangleq \mathtt{F}^{\mathtt{lwsync}}$$
$$(\!|\mathtt{W}^{o_{\mathtt{W}}}_{o_{\mathtt{RMW}}}(x, v)|\!) \triangleq \mathtt{W}(x, v) \qquad (\!|\mathtt{F}^{\mathtt{sc}}|\!) \triangleq \mathtt{F}^{\mathtt{sync}}$$

- $G.\mathtt{rmw} = G_p.\mathtt{rmw}$, $G.\mathtt{data} = G_p.\mathtt{data}$, and $G.\mathtt{addr} = G_p.\mathtt{addr}$
  (the compilation does not change RMW pairs and data/address dependencies)
- $G.\mathtt{ctrl} \subseteq G_p.\mathtt{ctrl}$
  (the compilation only adds control dependencies)
- $[G.\mathtt{R}^{\mathtt{acq}}] ; G.\mathtt{po} \subseteq G_p.\mathtt{rmw} \cup G_p.\mathtt{ctrl}$
  (a control dependency is placed from every acquire read)
- $[G.\mathtt{R}_{\mathtt{ex}}] ; G.\mathtt{po} \subseteq G_p.\mathtt{ctrl} \cup G_p.\mathtt{rmw} \cap G_p.\mathtt{data}$
  (exclusive reads entail a control dependency to any future event, except for their immediate exclusive write successor if arose from an atomic increment)
- $G.\mathtt{data} ; [\mathit{codom}(G.\mathtt{rmw})] ; G.\mathtt{po} \subseteq G_p.\mathtt{ctrl}$
  (data dependency to an exclusive write entails a control dependency to any future event)

[7] Since IMM does not have a primitive that corresponds to POWER's instruction fence, we cannot apply the same trick for acquire reads.

$$(\!|r := [e]^{\mathsf{rlx}}|\!) \approx \text{“}\texttt{ldr}\text{”} \qquad (\!|[e_1]^{\mathsf{rlx}} := e_2|\!) \approx \text{“}\texttt{str}\text{”}$$
$$(\!|r := [e]^{\mathsf{acq}}|\!) \approx \text{“}\texttt{ldar}\text{”} \qquad (\!|[e_1]^{\mathsf{rel}} := e_2|\!) \approx \text{“}\texttt{stlr}\text{”}$$
$$(\!|\mathbf{fence}^{\mathsf{acq}}|\!) \approx \text{“}\texttt{dmb.ld}\text{”} \qquad (\!|\mathbf{fence}^{\neq\mathsf{acq}}|\!) \approx \text{“}\texttt{dmb.sy}\text{”}$$
$$(\!|r := \mathbf{FADD}^{o_{\mathsf{R}},o_{\mathsf{W}}}_{o_{\mathsf{RMW}}}(e_1,e_2)|\!) \approx \text{“L:”} +\!\!+ \mathrm{ld}(o_{\mathsf{R}}) +\!\!+ \mathrm{st}(o_{\mathsf{W}}) +\!\!+ \text{“}\texttt{bc L}\text{”} +\!\!+ \mathrm{dmb}(o_{\mathsf{RMW}})$$
$$(\!|r := \mathbf{CAS}^{o_{\mathsf{R}},o_{\mathsf{W}}}_{o_{\mathsf{RMW}}}(e,e_{\mathsf{R}},e_{\mathsf{W}})|\!) \approx \text{“L:”} +\!\!+ \mathrm{ld}(o_{\mathsf{R}}) +\!\!+ \text{“}\texttt{cmp;bc Le;}\text{”} +\!\!+ \mathrm{st}(o_{\mathsf{W}}) +\!\!+ \text{“}\texttt{bc L;Le:}\text{”} +\!\!+ \mathrm{dmb}(o_{\mathsf{RMW}})$$
$$\mathrm{ld}(o_{\mathsf{R}}) \triangleq o_{\mathsf{R}} = \mathsf{acq}\ ?\ \text{“}\texttt{ldaxr;}\text{”}\ :\ \text{“}\texttt{ldxr;}\text{”} \qquad \mathrm{st}(o_{\mathsf{W}}) \triangleq o_{\mathsf{W}} = \mathtt{rel}\ ?\ \text{“}\texttt{stlxr.;}\text{”}\ :\ \text{“}\texttt{stxr.;}\text{”}$$
$$\mathrm{dmb}(o_{\mathsf{RMW}}) \triangleq o_{\mathsf{RMW}} = \mathtt{strong}\ ?\ \text{“}\texttt{;dmb.ld}\text{”}\ :\ \text{“”}$$

Fig. 5. Compilation scheme from IMM to ARMv8.

- $G.\mathsf{casdep}\,;G.\mathsf{po} \subseteq G_p.\mathtt{ctrl}$
  (CAS dependency to an exclusive read entails a control dependency to any future event)

Next, we state our theorem that ensures IMM-consistency if the corresponding POWER execution graph is POWER-consistent. Due to lack of space, we do not include here the (quite elaborate) definition of POWER-consistency. For that definition, we refer the reader to [Alglave et al. 2014] ([Podkopaev et al. 2018, Appendix B] provides the definition we used in our development).

Theorem 4.3. *Let G be an* IMM *execution graph with whole serial numbers ($\mathsf{sn}[G.\mathsf{E}] \subseteq \mathbb{N}$), such that $G.\mathtt{W}^{\mathtt{rel}} = \emptyset$, and let G_p be a* POWER *execution graph that corresponds to G. Then,* POWER*-consistency of G_p implies* IMM*-consistency of G.*

The ARMv7 model in [Alglave et al. 2014] is very similar to the POWER model. There are only two differences. First, ARMv7 lacks an analogue for POWER's lightweight fence (`lwsync`). Second, ARMv7 has a weaker preserved program order than POWER, which in particular does not always include $[G.\mathtt{R}];G.\mathsf{po}|_{G.\mathtt{loc}};[G.\mathtt{W}]$ (the $\mathsf{po}|_{\mathtt{loc}}/\mathtt{cc}$ rule is excluded, see [Podkopaev et al. 2018,)]. In our proofs for POWER, however, we never rely on POWER's ppo, but rather assume the weaker one of ARMv7. The compilation schemes to ARMv7 are essentially the same as those to POWER substituting the corresponding ARMv7 instructions for the POWER ones: `dmb` instead of `sync` and `lwsync`, and `isb` instead of `isync`. Thus, the correctness of compilation to ARMv7 follows directly from the correctness of compilation to POWER.

4.2 From IMM to ARMv8

The intended mapping of IMM to ARMv8 is presented schematically in Fig. 5. It is identical to the mapping to POWER (Fig. 4), except for the following:

- Unlike POWER, ARMv8 has machine instructions for acquire loads (`ldar`) and release stores (`stlr`), which are used instead of placing barriers next to plain loads and stores.
- ARMv8 has a special `dmb.ld` barrier that is used for IMM's acquire fences. On the other side, it lacks an analogue for IMM's release fence, for which a full barrier (`dmb.sy`) is used.
- As noted in Example 3.10, the mapping of IMM's *strong* RMWs requires placing a `dmb.ld` barrier after the exclusive write.

As a model of the ARMv8 architecture, we use its recent official declarative model [Deacon 2017] (see also [Pulte et al. 2018]) which we denote by ARM.[8] Its labels are given by:

- ARM read label: $\mathtt{R}^{o_{\mathsf{R}}}(x,v)$ where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, and $o_{\mathsf{R}} \in \{\mathtt{rlx},\mathtt{Q}\}$.
- ARM write label: $\mathtt{W}^{o_{\mathsf{W}}}(x,v)$ where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, and $o_{\mathsf{W}} \in \{\mathtt{rlx},\mathtt{L}\}$.
- ARM fence label: $\mathtt{F}^{o_{\mathsf{F}}}$ where $o_{\mathsf{F}} \in \{\mathtt{ld},\mathtt{sy}\}$.

[8] We only describe the fragment of the model that is needed for mapping of IMM, thus excluding sequentially consistent reads and `isb` fences.

In turn, ARM's execution graphs are defined as IMM's ones, except for the CAS dependency, casdep, which is not present in ARM executions. As we did for POWER, we first interpret the intended compilation on execution graphs:

Definition 4.4. Let G be an IMM execution graph with whole serial numbers ($\mathsf{sn}[G.\mathsf{E}] \subseteq \mathbb{N}$). An ARM execution graph G_a *corresponds* to G if the following hold (we skip the explanation of conditions that appear in Def. 4.2):

- $G_a.\mathsf{E} = G.\mathsf{E} \cup \{\langle i, n+0.5\rangle \mid \langle i, n\rangle \in G.\mathsf{W}_{\mathsf{strong}}\}$
  (new events are added after strong exclusive writes)
- $G_a.\mathtt{lab} = \{e \mapsto (\!|G.\mathtt{lab}(e)|\!) \mid e \in G.\mathsf{E}\} \cup \{e \mapsto \mathsf{F}^{\mathsf{ld}} \mid e \in G_a.\mathsf{E} \setminus G.\mathsf{E}\}$ where:

$$(\!|\mathsf{R}_s^{\mathsf{rlx}}(x,v)|\!) \triangleq \mathsf{R}^{\mathsf{rlx}}(x,v) \qquad (\!|\mathsf{W}_{o_{\mathsf{RMW}}}^{\mathsf{rlx}}(x,v)|\!) \triangleq \mathsf{W}^{\mathsf{rlx}}(x,v)$$
$$(\!|\mathsf{R}_s^{\mathsf{acq}}(x,v)|\!) \triangleq \mathsf{R}^{\mathsf{Q}}(x,v) \qquad (\!|\mathsf{W}_{o_{\mathsf{RMW}}}^{\mathsf{rel}}(x,v)|\!) \triangleq \mathsf{W}^{\mathsf{L}}(x,v)$$
$$(\!|\mathsf{F}^{\mathsf{acq}}|\!) \triangleq \mathsf{F}^{\mathsf{ld}} \qquad (\!|\mathsf{F}^{\mathsf{rel}}|\!) = (\!|\mathsf{F}^{\mathsf{acqrel}}|\!) = (\!|\mathsf{F}^{\mathsf{sc}}|\!) \triangleq \mathsf{F}^{\mathsf{sy}}$$

- $G.\mathtt{rmw} = G_a.\mathtt{rmw}$, $G.\mathtt{data} = G_a.\mathtt{data}$, and $G.\mathtt{addr} = G_a.\mathtt{addr}$
- $G.\mathtt{ctrl} \subseteq G_a.\mathtt{ctrl}$
- $[G.\mathsf{R}_{\mathsf{ex}}]\,;G.\mathsf{po} \subseteq G_a.\mathtt{ctrl} \cup G_a.\mathtt{rmw} \cap G_a.\mathtt{data}$
- $G.\mathtt{casdep}\,;G.\mathsf{po} \subseteq G_a.\mathtt{ctrl}$

Next, we state our theorem that ensures IMM-consistency if the corresponding ARM execution graph is ARM-consistent. Again, due to lack of space, we do not include here the definition of ARM-consistency. For that definition, we refer the reader to [Deacon 2017; Pulte et al. 2018] ([Podkopaev et al. 2018, Appendix C] provides the definition we used in our development).

Theorem 4.5. *Let G be an* IMM *execution graph with whole serial numbers ($\mathsf{sn}[G.\mathsf{E}] \subseteq \mathbb{N}$), and let G_a be an* ARM *execution graph that corresponds to G. Then,* ARM*-consistency of G_a implies* IMM*-consistency of G.*

5 FROM C11 AND RC11 TO IMM

In this section, we establish the correctness of the mapping from the C11 and RC11 models to IMM. Since C11 and RC11 are defined declaratively and IMM-consistency is very close to (R)C11-consistency, these results are straightforward.

Incorporating the fixes from Vafeiadis et al. [2015] and Lahav et al. [2017] to the original C11 model of Batty et al. [2011], and restricting attention to the fragment of C11 that has direct IMM counterparts (thus, excluding non-atomic and SC accesses), C11-consistency is defined follows.

Definition 5.1. G is called *C11-consistent* if the following hold:

- $\mathit{codom}(G.\mathtt{rf}) = G.\mathsf{R}$.
- For every location $x \in \mathsf{Loc}$, $G.\mathtt{co}$ totally orders $G.\mathsf{W}(x)$.
- $G.\mathsf{hb}_{\mathrm{RC11}}\,;G.\mathtt{eco}^?$ is irreflexive.
- $G.\mathtt{rmw} \cap (G.\mathtt{fre}\,;G.\mathtt{coe}) = \emptyset$.
- $[\mathsf{F}^{\mathsf{sc}}]\,;(\mathsf{hb}_{\mathrm{RC11}} \cup \mathsf{hb}_{\mathrm{RC11}}\,;\mathtt{eco}\,;\mathsf{hb}_{\mathrm{RC11}})\,;[\mathsf{F}^{\mathsf{sc}}]$ is acyclic.

It is easy to show that IMM-consistency implies C11-consistency, and consequently, the identity mapping is a correct compilation from this fragment of C11 to IMM. This result can be extended to include non-atomic and SC accesses as follows:

- Non-atomic accesses provide weaker guarantees than relaxed accesses, and are not needed for accounting for IMM's behaviors. Put differently, one may assume that the compilation from C11 to IMM first strengthens all non-atomic accesses to relaxed accesses. Compilation correctness then follows from the soundness of this strengthening and our result that excludes non-atomics.

- The semantics of SC accesses in C11 was shown to be too strong in [Lahav et al. 2017; Manerkar et al. 2016] to allow the intended compilation to POWER and ARMv7. If one applies the fix proposed in [Lahav et al. 2017], then compilation correctness could be established following their reduction, that showed that it is sound to globally split SC accesses to SC fences and release/acquire accesses on the source level. This encoding yields the (two) expected compilation schemes for SC loads and stores on x86, ARMv7, and POWER. On the other hand, handling ARMv8's specific instructions for SC accesses is left for future work. We note that the usefulness and the "right semantics" for SC accesses is still under discussion. The Promising semantics, for instance, does not have primitive SC accesses at all and implements them using SC fences.

In turn, RC11 (ignoring the part related to SC accesses) is obtained by strengthening Def. 5.1 with a condition asserting that $G.\mathsf{po} \cup G.\mathtt{rf}$ is acyclic. To enforce the additional requirement, the mapping of RC11 places a (control) dependency or a fence between every relaxed read and subsequent relaxed write. It is then straightforward to define the correspondence between source (RC11) execution graphs and target (IMM) ones, and prove that IMM-consistency of the target graph implies RC11-consistency of the source. This establishes the correctness of the intended mapping from RC11 without non-atomic accesses to IMM. Handling non-atomic accesses, which are intended to be mapped to plain machine accesses with no additional barriers or dependencies (on which IMM generally allows $\mathsf{po} \cup \mathtt{rf}$-cycles), is left for future work; while SC accesses can be handled as mentioned above.

6 FROM THE PROMISING SEMANTICS TO IMM: RELAXED FRAGMENT

In the section, we outline the main ideas of the proof of the correctness of compilation from the promising semantics of Kang et al. [2017], denoted by Promise, to IMM. To assist the reader, we initially restrict attention to programs containing only relaxed read and write accesses. In §7, we show how to adapt and extend our proof to the full model.

Our goal is to prove that for every outcome of a program *prog* (with relaxed accesses only) under IMM (Def. 2.9), there exists a Promise trace of *prog* terminating with the same outcome. To do so, we introduce a traversal strategy of IMM-consistent execution graphs, and show, by forward simulation argument, that it can be followed by Promise. The main challenge in the simulation proof is due to the *certification* requirement of Promise—after every step, the thread that made the transition has to show that it can run in isolation and fulfill all its so-called promises. To address this challenge, we break our simulation argument into two parts. First, we provide a simulation relation, which relates a Promise thread state with a traversal configuration. Second, after each traversal step, we (i) construct a *certification execution graph* G^{crt} and a new traversal configuration TC^{crt}; (ii) show that the simulation relation relates G^{crt}, TC^{crt}, and the current Promise state; and (iii) deduce that we can meet the certification condition by traversing G^{crt}. (Here, we use the fact that Promise does not require nested certifications.)

The rest of this section is structured as follows. In §6.1 we describe the fragment of Promise restricted to relaxed accesses. In §6.2 we introduce the traversal of IMM-consistent execution graphs, which is suitable for the relaxed fragment. In §6.3 we define the simulation relation for Promise thread steps and the execution graph traversal. In §6.4 we discuss how we handle certification. Finally, in §6.5 we state the compilation correctness theorem and provide its proof outline.

6.1 The Promise Machine (Relaxed Fragment)

Promise is an operational model where threads execute in an interleaved fashion. The *machine state* is a pair $\Sigma = \langle \mathcal{TS}, M \rangle$, where $\mathcal{TS}$ assigns a *thread state* TS to every thread and M is a (global) *memory*. The memory consists of a set of *messages* of the form $\langle x : v@t \rangle$ representing all previously

executed writes, where $x \in \mathsf{Loc}$ is the target location, $v \in \mathsf{Val}$ is the stored value, and $t \in \mathbb{Q}$ is the *timestamp*. The timestamps totally order the messages to each location (this order corresponds to $G.\mathsf{co}$ in our simulation proof).

The state of each thread contains a *thread view*, $\mathcal{V} \in \mathsf{View} \triangleq \mathsf{Loc} \to \mathbb{Q}$, which represents the "knowledge" of each thread. The view is used to forbid a thread to read from a (stale) message $\langle x : v@t \rangle$ if it is aware of a newer one, *i.e.,* when $\mathcal{V}(x)$ is greater than t. Also, it disallows to write a message to the memory with a timestamp not greater than $\mathcal{V}(x)$. (Due to lack of space, we refer the reader to Kang et al. [2017] for the full definition of thread steps.)

Besides the step-by-step execution of their programs, threads may non-deterministically *promise* future writes. This is done by simply adding a message to the memory. We refer to the execution of a write instruction whose message was promised before as *fulfilling* the promise.

The *thread state* TS is a triple $\langle \sigma, \mathcal{V}, P \rangle$, where σ is the thread's local state,[9] $\mathcal{V}$ is the thread view, and P tracks the set of messages that were promised by the thread and not yet fulfilled. We write $TS.\mathtt{prm}$ to obtain the promise set of a thread state TS. Initially, each thread is in local state $TS_0^i = \langle \sigma_0(prog(i)), \lambda x.\ 0, \emptyset \rangle$.

To ensure that promises do not make the semantics overly weak, each sequence of thread steps in Promise has to be *certified*: the thread that took the steps should be able to fulfill all its promises when executed in isolation. Thus, a *machine step* in Promise is given by:

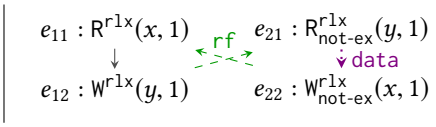
$$\frac{\langle \mathcal{TS}(i), M \rangle \to^+ \langle TS', M' \rangle \qquad \exists TS''.\ \langle TS', M' \rangle \to^* \langle TS'', \_ \rangle \wedge TS''.\mathtt{prm} = \emptyset}{\langle \mathcal{TS}, M \rangle \to \langle \mathcal{TS}[i \mapsto TS'], M' \rangle}$$

Program outcomes under Promise are defined as follows.

Definition 6.1. A function $O : \mathsf{Loc} \to \mathsf{Val}$ is an *outcome* of a program $prog$ under Promise if $\Sigma_0(prog) \to^* \langle \mathcal{TS}, M \rangle$ for some $\mathcal{TS}$ and M such that the thread's local state in $\mathcal{TS}(i)$ is terminal for every $i \in \mathsf{Tid}$, and for every $x \in \mathsf{Loc}$, there exists a message of the form $\langle x : O(x)@t \rangle \in M$ where t is maximal among timestamps of messages to x in M. Here, $\Sigma_0(prog)$ denotes the initial machine state, $\langle \mathcal{TS}_{\mathsf{init}}, M_{\mathsf{init}} \rangle$, where $\mathcal{TS}_{\mathsf{init}} = \lambda i.\ TS_0^i$, and $M_{\mathsf{init}} = \{\langle x : 0@0 \rangle \mid x \in \mathsf{Loc}\}$.

Example 6.2 (Load Buffering). Consider the following load buffering behavior under IMM:

$$\begin{array}{l} e_{11} : a := [x]^{\mathtt{rlx}} \ \#\!/\ 1 \\ e_{12} : [y]^{\mathtt{rlx}} := 1 \end{array} \Bigg\| \begin{array}{l} e_{21} : b := [y]^{\mathtt{rlx}} \ \#\!/\ 1 \\ e_{22} : [x]^{\mathtt{rlx}} := b \end{array}$$



The Promise machine obtains this outcome as follows. Starting with memory $\langle \langle x : 0@0 \rangle, \langle y : 0@0 \rangle \rangle$, the left thread promises the message $\langle y : 1@1 \rangle$. After that, the right thread reads this message and executes its second instruction (promises a write and immediately fulfills it), adding the the message $\langle x : 1@1 \rangle$ to memory. Then, the left thread reads from that message and fulfills its promise. Each step (including, in particular, the first promise step) could be easily "certified" in a thread-local execution. Note also how the data dependency in the right thread redistrict the execution of the Promise machine. Due to the certification requirement, the execution cannot begin by the right thread promising $\langle x : 1@1 \rangle$, as it cannot generate this message by running in isolation. □

6.2 Traversal (Relaxed Fragment)

Our goal is to generate a run of Promise for any given IMM-consistent initialized execution graph G of a program $prog$. To do so, we traverse G with a certain strategy, deciding in each step whether

[9] The promising semantics is generally formulated over a general labeled state transition system. In our development, we instantiate it with the sequential program semantics that is used in §2.3 to construct execution graphs.

to execute the next instruction in the program or promise a future write. While traversing G, we keep track of a *traversal configuration*—a pair $TC = \langle C, I\rangle$ of subsets of $G.\mathsf{E}$. We call the events in C and I *covered* and *issued* respectively. The covered events correspond to the instructions that were executed by Promise, and the issued events corresponds to messages that were added to the memory (executed or promised stores).

Initially, we take $TC_0 = \langle G.\mathsf{E} \cap \mathsf{Init}, G.\mathsf{E} \cap \mathsf{Init}\rangle$. Then, at each traversal step, the covered and/or issued sets are increased, using one of the following two steps:

$$\frac{w \in \mathsf{Issuable}(G, C, I)}{G \vdash \langle C, I\rangle \to_{\mathtt{tid}(w)} \langle C, I \uplus \{w\}\rangle}\ (\textsc{issue}) \qquad \frac{e \in \mathsf{Coverable}(G, C, I)}{G \vdash \langle C, I\rangle \to_{\mathtt{tid}(e)} \langle C \uplus \{e\}, I\rangle}\ (\textsc{cover})$$

The (ISSUE) step adds an event w to the issued set. It corresponds to a promise step of Promise. We require that w is issuable, which says that all the writes of other threads that it depends on have already been issued:

Definition 6.3. An event w is *issuable* in G and $\langle C, I\rangle$, denoted $w \in \mathsf{Issuable}(G, C, I)$, if $w \in G.\mathtt{W}$ and $dom(G.\mathtt{rfe} ; G.\mathtt{ppo} ; [w]) \subseteq I$.

The (COVER) step adds an event e to the covered set. It corresponds to an execution of a program instruction in Promise. We require that e is coverable, as defined next.

Definition 6.4. An event e is called *coverable* in G and $\langle C, I\rangle$, denoted $e \in \mathsf{Coverable}(G, C, I)$, if $e \in G.\mathsf{E}$, $dom(G.\mathtt{po} ; [e]) \subseteq C$, and either (i) $e \in G.\mathtt{W} \cap I$; or (ii) $e \in G.\mathsf{R}$ and $dom(G.\mathtt{rf} ; [e]) \subseteq I$.

The requirements in this definition are straightforward. First, all $G.\mathtt{po}$-previous events have to be covered, *i.e.,* previous instructions have to be already executed by Promise. Second, if e is a write event, then it has to be already issued; and if e is a read event, then the write event that e reads from has to be already issued (the corresponding message has to be available in the memory).

As an example of a traversal, consider the execution from Example 6.2. A possible traversal of the execution is the following: issue e_{12}, cover e_{21}, issue e_{22}, cover e_{22}, cover e_{11}, and cover e_{12}.

Starting from the initial configuration TC_0, each traversal step maintains the following invariants: (i) $\mathsf{E} \cap \mathsf{Init} \subseteq C$; (ii) $C \cap G.\mathtt{W} \subseteq I$; and (iii) $I \subseteq \mathsf{Issuable}(G, C, I)$ and $C \subseteq \mathsf{Coverable}(G, C, I)$. When these properties hold, we say that $\langle C, I\rangle$ is a *traversal configuration* of G. The next proposition ensures the existence of a traversal starting from any traversal configuration. (A proof outline for an extended version of the traversal discussed in §7.2 is presented in [Podkopaev et al. 2018, Appendix F].)

PROPOSITION 6.5. *Let G be an* IMM*-consistent execution graph and $\langle C, I\rangle$ be a traversal configuration of G. Then, $G \vdash \langle C, I\rangle \to^* \langle G.\mathsf{E}, G.\mathtt{W}\rangle$.*

6.3 Thread Step Simulation (Relaxed Fragment)

To show that a traversal step of thread i can be matched by a Promise thread step, we use a simulation relation $\mathcal{I}_i(G, TC, \langle TS, M\rangle, T)$, where G is an IMM-consistent initialized full execution of *prog*; $TC = \langle C, I\rangle$ is a traversal configuration of G; $TS = \langle \sigma, \mathcal{V}, P\rangle$ is i's thread state in Promise; M is the memory of Promise; and $T : I \to \mathbb{Q}$ is a function that assigns timestamps to issued writes. The relation $\mathcal{I}_i(G, TC, \langle TS, M\rangle, T)$ holds if the following conditions are met (for conciseness we omit the "$G.$" prefix):

(1) T agrees with $\mathtt{co}$:
- $\forall w \in \mathsf{E} \cap \mathsf{Init}.\ T(w) = 0$
- $\forall \langle w, w'\rangle \in [I] ; \mathtt{co} ; [I].\ T(w) \le T(w')$

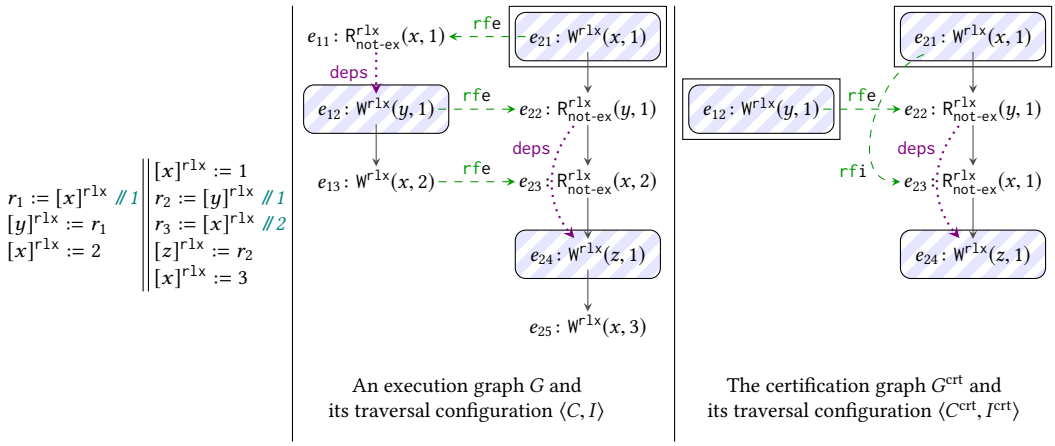


Fig. 6. A program, its execution graph, and a related certification graph. Covered events are marked by □ and issued ones by ▢.

(2) Non-initialization messages in M have counterparts in I:
- $\forall\langle x : \_@t\rangle \in M.\ t \neq 0 \Rightarrow \exists w \in I.\ \mathtt{loc}(w) = x \wedge T(w) = t$

(3) Issued events have corresponding messages in memory:
- $\forall w \in I.\ \langle \mathtt{loc}(w) : \mathtt{val}(w)@T(w)\rangle \in M$

(4) For every promise, there exists a corresponding issued uncovered event w:
- $\forall\langle x : v@t\rangle \in P.\ \exists w \in \mathsf{E}_i \cap I \setminus C.\ \mathtt{loc}(w) = x \wedge \mathtt{val}(w) = v \wedge T(w) = t$

(5) Every issued uncovered event w of thread i has a corresponding promise in P.
- $\forall w \in \mathsf{E}_i \cap I \setminus C.\ \langle \mathtt{loc}(w) : \mathtt{val}(w)@T(w)\rangle \in P$

(6) The view $\mathcal{V}$ is justified by graph paths:
- $\mathcal{V} = \lambda x.\ \max T[\mathtt{W}(x) \cap dom(\mathtt{vf}_{\mathtt{rlx}}\ ;\ [\mathsf{E}_i \cap C])]$ where $\mathtt{vf}_{\mathtt{rlx}} \triangleq \mathtt{rf}^?; \mathsf{po}^?$

(7) The thread local state σ matches the covered events ($\sigma.\mathtt{G.E} = C \cap \mathsf{E}_i$), and can always reach the execution graph G ($\exists\sigma'.\ \sigma \to_i^* \sigma' \wedge \sigma'.\mathtt{G} = G|_i$).

Proposition 6.6. *If $\mathcal{I}_i(G, TC, \langle TS, M\rangle, T)$ and $G \vdash TC \to_i TC'$ hold, then there exist TS', M', T' such that $\langle TS, M\rangle \to \langle TS', M'\rangle$ and $\mathcal{I}_i(G, TC', \langle TS', M'\rangle, T')$ hold.*

In addition, it is easy to verify that the initial states are related, *i.e.,* $\mathcal{I}_i(G, TC_0, \langle TS_0^i, M_{\mathrm{init}}\rangle, \bot)$ holds for every $i \in \mathrm{Tid}$.

6.4 Certification (Relaxed Fragment)

To show that a traversal step can be simulated by Promise, Prop. 6.6 does not suffice: the machine step requires the new thread's state to be certified. To understand how we construct a certification run, consider the example in Fig. 6. Suppose that $\mathcal{I}_{i_2}$ holds for $G, \langle C, I\rangle, \langle TS, M\rangle, T$ (where i_2 is the identifier of the second thread). Consider a possible certification run for i_2. According to $\mathcal{I}_{i_2}$, there is one unfulfilled promise of i_2, *i.e., TS.*$\mathtt{prm} = \{\langle z : 1@T(e_{24})\rangle\}$. We also know that i_2 has executed all instructions up to the one related to the last covered event e_{21}. To fulfill the promise, it has to execute the instructions corresponding to e_{22}, e_{23}, and e_{24}.

To construct the certification run, we (inductively) apply a version of Prop. 6.6 for certification steps, starting from a sequence of traversal steps of i_2 that cover e_{22}, e_{23}, and e_{24}. For G and $\langle C, I\rangle$, there is no such sequence: we cannot cover e_{23} without issuing e_{13} first (which we cannot do since only one thread may run during certification). Nevertheless, observing that the value read at e_{23} is immaterial for covering e_{24}, we may use a different execution graph for this run, namely G^{crt}

shown in Fig. 6. Thus, in G^{crt} we redirect e_{23}'s incoming reads-from edge and change its value accordingly. In contrast, we do not need to change e_{22}'s incoming reads-from edge because the condition about $G.\mathsf{ppo}$ in the definition of issuable events ensures that e_{12} must have already been issued. For G^{crt} and $\langle C^{\text{crt}}, I^{\text{crt}}\rangle$, there exists a sequence of traversal steps that cover e_{22}, e_{23}, and e_{24}. Since events of other threads have all been made covered in C^{crt}, we know that only i_2 will take steps in this sequence.

Generally speaking, for a given $i \in \mathsf{Tid}$ whose step has to be certified, our goal is to construct a certification graph G^{crt} and a traversal configuration $TC^{\text{crt}} = \langle C^{\text{crt}}, I^{\text{crt}}\rangle$ of G^{crt} such that (1) G^{crt} is IMM-consistent (so we can apply Prop. 6.5 to it) and (2) we can simulate its traversal in Promise to obtain the certification run for thread i. In particular, the latter requires that $G^{\text{crt}}|_i$ is an execution graph of i's program. In the rest of this section, we present this construction and show how it is used to certify Promise's steps (Prop. 6.9).

First, the events of G^{crt} are given by $G^{\text{crt}}.\mathsf{E} \triangleq C \cup I \cup \mathit{dom}(G.\mathsf{po}\,;[I \cap G.\mathsf{E}_i])$. They consist of the covered and issued events and all po-preceding events of issued events in thread i. The co and dependency components of G^{crt} are the same as in (restricted) G ($G^{\text{crt}}.\mathsf{x} = [G^{\text{crt}}.\mathsf{E}]\,;G.\mathsf{x}\,;[G^{\text{crt}}.\mathsf{E}]$ for $\mathsf{x} \in \{\mathsf{co}, \mathsf{addr}, \mathsf{data}, \mathsf{ctrl}, \mathsf{casdep}\}$). As we saw on Fig. 6, we may need to modify the rf edges of the certification graph (and, consequentially, labels of events). In the example, it was required because the source of an rf edge was not present in G^{crt}. The relation $G^{\text{crt}}.\mathsf{rf}$ is defined as follows:

$$G^{\text{crt}}.\mathsf{rf} \triangleq G.\mathsf{rf}\,;[D] \cup \bigcup_{x\in\mathsf{Loc}}([G.\mathsf{W}(x)]\,;\mathsf{bvf}^{\mathsf{rlx}}\,;[G.\mathsf{R}(x) \cap G^{\text{crt}}.\mathsf{E} \setminus D] \setminus G.\mathsf{co}\,;G.\mathsf{bvf}^{\mathsf{rlx}})$$
$$\text{where } D = G^{\text{crt}}.\mathsf{E} \cap (C \cup I \cup G.\mathsf{E}_{\neq i} \cup \mathit{dom}(G.\mathsf{rfi}^{?}\,;G.\mathsf{ppo}\,;[I])) \text{ and}$$
$$G.\mathsf{bvf}^{\mathsf{rlx}} = (G.\mathsf{rf}\,;[D])^{?}\,;G.\mathsf{po}$$

The set D represents the *determined* events, whose rf edges are preserved. Intuitively, for a read event r with location x, the set $\mathit{dom}([G.\mathsf{W}(x)]\,;G.\mathsf{bvf}^{\mathsf{rlx}}\,;[r])$ consists of writes to x that are "observed" by $\mathsf{tid}(r)$ at the moment it "executes" r. If r is not determined, we choose the new rf edge to r to be from the co-latest write in this set. Thus, in the certification graph, r is not reading a stale value, and its incoming rf edge does not increase the set of "observed" writes in thread i.

The labels (which include the read values) in G^{crt} have to be modified as well, to match the new rf edges. To construct of $G^{\text{crt}}.\mathsf{lab}$, we leverage a certain *receptiveness* property of the operational semantics in Fig. 3. Roughly speaking, we show that if $\langle \mathit{sprog}, \mathit{pc}, \Phi, G, \Psi, S\rangle \to_i^+ \langle \mathit{sprog}, \mathit{pc}', \Phi', G', \Psi', S'\rangle$, then for every read $r \in G'.\mathsf{E} \setminus (G.\mathsf{E} \cup \mathit{dom}(G'.\mathsf{ctrl}))$ and value v, there exist pc'', Φ'', G'', Ψ'', and S'' such that $\langle \mathit{sprog}, \mathit{pc}, \Phi, G, \Psi, S\rangle \to_i^+ \langle \mathit{sprog}, \mathit{pc}'', \Phi'', G'', \Psi'', S''\rangle$, $G''.\mathsf{val}(r) = v$, and G'' is identical to G' except (possibly) for values of events that depend on r.[10] Applying this property inductively, we construct the labeling function $G^{\text{crt}}.\mathsf{lab}$.

This concludes the construction of G^{crt}. Now, we start the traversal from $TC^{\text{crt}} = \langle C^{\text{crt}}, I^{\text{crt}}\rangle$ where $C^{\text{crt}} \triangleq C \cup G^{\text{crt}}.\mathsf{E}_{\neq i}$ and $I^{\text{crt}} \triangleq I$. Thus, we take all events of other threads to be covered so that the traversal of G^{crt} may only include steps of thread i. To be able to reuse Prop. 6.5, we prove the following proposition.

Proposition 6.7. *Let G be an IMM-consistent execution graph, and $TC = \langle C, I\rangle$ a traversal configuration of G. Then, G^{crt} is IMM-consistent and TC^{crt} is a traversal configuration of G^{crt}.*

For the full model (see §7.4), we will have to introduce a slightly modified version of the simulation relation for certification. For the relaxed fragment that we consider here, however, we use the same relation defined in §6.3 and prove that it holds for the constructed certification graph:

Proposition 6.8. *Suppose that $\mathcal{I}_i(G, TC, \langle TS, M\rangle, T)$ holds. Then $\mathcal{I}_i(G^{\text{crt}}, TC^{\text{crt}}, \langle TS, M\rangle, T)$ holds.*

[10] The full formulation of the receptiveness property is more elaborate. Due to the lack of space, we refer the reader to our Coq development [Podkopaev et al. 2018].

Putting Prop. 6.5 to 6.8 together, we derive the following strengthened version of Prop. 6.6, which additionally states that the new Promise thread's state is certifiable.

Proposition 6.9. *If $\mathcal{I}_i(G, TC, \langle TS, M\rangle, T)$ and $G \vdash TC \rightarrow_i TC'$ hold, then there exist TS', M', T' such that $\langle TS, M\rangle \rightarrow^+ \langle TS', M'\rangle$ and $\mathcal{I}_i(G, TC', \langle TS', M'\rangle, T')$ hold, and there exist TS'', M'' such that $\langle TS', M'\rangle \rightarrow^* \langle TS'', M''\rangle$ and $TS''.\mathtt{prm} = \emptyset$.*

Proof outline. By Prop. 6.6, there exist TS', M', and T' such that $\langle TS, M\rangle \rightarrow^+ \langle TS', M'\rangle$ and $\mathcal{I}_i(G, TC', \langle TS', M'\rangle, T')$ hold. By Prop. 6.8, $\mathcal{I}_i(G^{\mathrm{crt}}, TC^{\mathrm{crt}}, \langle TS', M'\rangle, T')$ holds. By Prop. 6.5 and 6.7, we have $G^{\mathrm{crt}} \vdash TC^{\mathrm{crt}} \rightarrow_i^* \langle G^{\mathrm{crt}}.\mathsf{E}, G^{\mathrm{crt}}.\mathtt{W}\rangle$. We inductively apply Prop. 6.6 to obtain $\langle TS'', M''\rangle$ and T'' such that $\langle TS', M''\rangle \rightarrow^* \langle TS'', M''\rangle$ and $\mathcal{I}_i(G^{\mathrm{crt}}, \langle G^{\mathrm{crt}}.\mathsf{E}, G^{\mathrm{crt}}.\mathtt{W}\rangle, \langle TS'', M''\rangle, T'')$ hold. From the latter, it follows that $TS''.\mathtt{prm} = \emptyset$. □

6.5 Compilation Correctness Theorem (Relaxed Fragment)

Theorem 6.10. *Let prog be a program with only relaxed reads and relaxed writes. Then, every outcome of prog under* IMM *(Def. 2.9) is also an outcome of prog under* Promise *(Def. 6.1).*

Proof outline. We introduce a simulation relation $\mathcal{J}$ on traversal configurations and Promise states:

$$\mathcal{J}(G, TC, \langle \mathcal{TS}, M\rangle, T) \triangleq \forall i \in \mathsf{Tid}.\ \mathcal{I}_i(G, TC, \langle \mathcal{TS}(i), M\rangle, T)$$

We show that $\mathcal{J}$ holds for an IMM-consistent execution graph G, which has the outcome O, of the program *prog*, its initial traversal configuration, the initial Promise state $\Sigma_0(prog)$, and the initial timestamp mapping $T = \bot$. Then, we inductively apply Prop. 6.9 on a traversal $G \vdash \langle G.\mathsf{E} \cap \mathsf{Init}, G.\mathsf{E} \cap \mathsf{Init}\rangle \rightarrow^* \langle G.\mathsf{E}, G.\mathtt{W}\rangle$, which exists by Prop. 6.5, and additionally show that at every step $\mathcal{I}_i$ holds for every thread i that did not take the step. Thus, we obtain a Promise state Σ and a timestamp function T such that $\Sigma_0(prog) \rightarrow^* \Sigma$ and $\mathcal{J}(G, \langle G.\mathsf{E}, G.\mathtt{W}\rangle, \Sigma, T)$ hold. From the latter, it follows that O is an outcome of *prog* under Promise. □

7 FROM THE PROMISING SEMANTICS TO IMM: THE GENERAL CASE

In the section, we extend the result of §6 to the full Promise model. Recall that, due to the limitation of Promise discussed in Example 3.10, we assume that all RMWs are "strong".

Theorem 7.1. *Let prog be a program in which all RMWs are "strong". Then, every outcome of prog under* IMM *is also an outcome of prog under* Promise.

To prove this theorem, we find it technically convenient to use a slightly modified version of IMM, which is (provably) weaker. In this version, we use the simplified synchronization relation $G.\mathtt{sw}_{\mathrm{RC11}}$ (see Remark 2), as well as a *total order on SC fences*, $G.\mathtt{sc}$, which we include as another basic component of execution graphs. Then, we include $G.\mathtt{sc}$ in $G.\mathtt{ar}$ instead of $G.\mathtt{psc}$ (see §3.3), and require that $G.\mathtt{sc}; G.\mathtt{hb}; (G.\mathtt{eco}; G.\mathtt{hb})^?$ is irreflexive (to ensure that $G.\mathtt{psc} \subseteq G.\mathtt{sc}$). It is easy to show that the latter modification results in an equivalent model, while the use of $G.\mathtt{sw}_{\mathrm{RC11}}$ makes this semantics only weaker than IMM. The $G.\mathtt{sc}$ relation facilitates the construction of a run of Promise, as it fully determines the order in which SC fences should be executed.

The rest of this section is structured as follows. In §7.1 we briefly introduce the full Promise model. In §7.2 we introduce more elaborate traversal of IMM execution graphs, which might be followed by the full Promise model. In §7.3 we define the simulation relation for the full model. In §7.4 we discuss how certification graphs are adapted for the full model.

7.1 The Full Promise Machine

In the full Promise model, the machine state is a triple $\Sigma = \langle \mathcal{TS}, \mathcal{S}, M\rangle$. The additional component $\mathcal{S} \in \mathsf{View}$ is a (global) *SC view*. Messages in the memory are of the form $\langle x : v@(f, t], view\rangle$,

where, comparing to the version from §6.1, (i) a timestamp t is extended to a *timestamp interval* $(f, t] \in \mathbb{Q} \times \mathbb{Q}$ satisfying $f < t$ or $f = t = 0$ (for initialization messages) and (ii) the additional component $view \in \mathsf{View}$ is the *message view*.[11] Messages to the same location should have disjoint timestamp intervals, and thus the intervals totally order the messages to each location. The use of intervals allows one to express the fact that two messages are adjacent (corresponding to $G.\mathsf{co}|_{\mathsf{imm}}$), which is required to enforce the RMW atomicity condition (§3.2).

Message views represent the "knowledge" carried by the message that is acquired by threads reading this message (if they use an acquire read or fence). In turn, the thread view $\mathcal{V}$ is now a triple $\langle \mathsf{cur}, \mathsf{acq}, \mathsf{rel} \rangle \in \mathsf{View} \times \mathsf{View} \times (\mathsf{Loc} \to \mathsf{View})$, whose components are called the *current*, *acquire*, and *release* views. The different thread steps (for the different program instructions) constrain the three components of the thread view with the timestamps and message views that are included in the messages that the thread reads and writes, as well as with the global SC view $\mathcal{S} \in \mathsf{View}$. These constraints are tailored to precisely enforce the coherence and RMW atomicity properties (§3.1,§3.2), as well as the global synchronization provided by SC fences. (Again, we refer the reader to Kang et al. [2017] for the full definition of thread steps.)

Apart from promising messages, our proof utilizes another non-deterministic step of Promise, which allows a thread to *split* its promised messages, *i.e.,* to replace its promise $\langle x : v@(f, t], view \rangle$ with two promises $\langle x : v'@(f, t'], view' \rangle$ and $\langle x : v@(t', t], view \rangle$ provided that $f < t' < t$.

In the full Promise model, the certification requirement is stronger than the one presented in §6 for the relaxed fragment. Due to possible interference of other threads before the current thread fulfills its promises, certification is required for every possible *future memory* and *future SC view*. Thus, a *machine step* in Promise is given by:

$$\frac{\langle \mathcal{TS}(i), \mathcal{S}, M \rangle \to^{+} \langle TS', \mathcal{S}', M' \rangle \qquad \forall M_{\mathrm{fut}} \supseteq M', \mathcal{S}_{\mathrm{fut}} \geq \mathcal{S}'.\ \exists TS''.\ \langle TS', \mathcal{S}_{\mathrm{fut}}, M_{\mathrm{fut}} \rangle \to^{*} \langle TS'', \_, \_ \rangle \wedge TS''.\mathtt{prm} = \emptyset}{\langle \mathcal{TS}, \mathcal{S}, M \rangle \to \langle \mathcal{TS}[i \mapsto TS'], \mathcal{S}', M' \rangle}$$

Example 7.2. We revisit the program presented in Example 3.6. To get the intended behavior in Promise, thread I starts by promising a message $\langle z : 1@(1, 2], [z@2] \rangle$. It may certify the promise since its fourth instruction does not depend on a and the thread may read 1 from y when executing the third instruction in any future memory. After the promise is added to memory, thread II reads it and writes $\langle x : 1@(1, 2], [x@2] \rangle$ to the memory. Then, thread I reads from this message, executes its remaining instructions, and fulfills its promise. □

Remark 3. In Promise, the notion of future memory is broader—a future memory may be obtained by a sequence of memory modifications including message additions, message splits and lowering of message views. In our Coq development, we show that it suffices to consider only future memories that are obtained by adding messages ([Podkopaev et al. 2018, Appendix E] outlines the proof of this claim).

Remark 4. What we outline here ignores Promise's *plain* accesses. These are weaker than relaxed accesses (they only provide partial coherence), and are not needed for accounting for IMM's behaviors. Put differently, one may assume that the compilation from Promise to IMM first strengthens all plain access modes to relaxed. The correctness of compilation then follows from the soundness of this strengthening (which was proved by Kang et al. [2017]) and our result that excludes plain accesses.

[11] The order $\leq$ on $\mathbb{Q}$ is extended pointwise to order $\mathsf{Loc} \to \mathbb{Q}$. $\bot$ and $\sqcup$ denote the natural bottom element and join operations (pointwise extensions of the initial timestamp 0 and the max operation on timestamps). $[x_1@t_1, \ldots, x_n@t_n]$ denotes the function assigning t_i to x_i and 0 to other locations.

$$\frac{w \in \mathsf{Issuable}(G, C, I) \qquad w \notin G.\mathtt{W}^{\mathtt{rel}}}{G \vdash \langle C, I\rangle \to_{\mathtt{tid}(w)} \langle C, I \uplus \{w\}\rangle}\ (\textsc{issue})$$

$$\frac{e \in \mathsf{Coverable}(G, C, I) \qquad e \notin \mathit{dom}(G.\mathtt{rmw})}{G \vdash \langle C, I\rangle \to_{\mathtt{tid}(e)} \langle C \uplus \{e\}, I\rangle}\ (\textsc{cover})$$

$$\frac{\mathit{dom}(G.\mathsf{po}\,;[w]) \subseteq C \qquad w \in G.\mathtt{W}^{\mathtt{rel}}}{G \vdash \langle C, I\rangle \to_{\mathtt{tid}(w)} \langle C \uplus \{w\}, I \uplus \{w\}\rangle}\ (\textsc{release-cover})$$

$$\frac{r \in \mathsf{Coverable}(G, C, I) \qquad \langle r, w\rangle \in G.\mathtt{rmw} \qquad (w \in I \wedge I' = I) \vee (w \in G.\mathtt{W}^{\mathtt{rel}} \wedge I' = I \uplus \{w\})}{G \vdash \langle C, I\rangle \to_{\mathtt{tid}(r)} \langle C \uplus \{r, w\}, I'\rangle}\ (\textsc{rmw-cover})$$

Fig. 7. Traversal steps.

7.2 Traversal

To support all features of IMM and Promise models, we have to complicate the traversal considered in §6.2. We do it by introducing two new traversal steps (see Fig. 7) and modifying the definitions of issuable and coverable events.

The (RELEASE-COVER) step is introduced because the Promise model forbids to promise a release write without fulfilling it immediately. It adds a release write to both the covered and issued sets in a single step. Its precondition is simple: all $G.\mathsf{po}$-previous events have to be covered.

The (RMW-COVER) step reflects that RMWs in Promise are performed in one atomic step, even though they are split to two events in IMM. Accordingly, when traversing G, we require to cover the write part of $\mathtt{rmw}$ edges immediately after their read part. If the write is release, then, again since release writes cannot be promised without immediate fulfillment, it is issued in the same step.

The full definition of issuable event has additional requirements.

Definition 7.3. An event w is *issuable* in G and $\langle C, I\rangle$, denoted $w \in \mathsf{Issuable}(G, C, I)$, if $w \in G.\mathtt{W}$ and the following hold:

- $\mathit{dom}(([G.\mathtt{W}^{\mathtt{rel}}]\,; G.\mathsf{po}|_{G.\mathtt{loc}} \cup [G.\mathtt{F}]\,; G.\mathsf{po})\,; [w]) \subseteq C$ (FWBOB-COV)
- $\mathit{dom}((G.\mathtt{detour} \cup G.\mathtt{rfe})\,; G.\mathsf{ppo}\,; [w]) \subseteq I$ (PPO-ISS)
- $\mathit{dom}((G.\mathtt{detour} \cup G.\mathtt{rfe})\,; [G.\mathtt{R}^{\mathtt{acq}}]\,; G.\mathsf{po}\,; [w]) \subseteq I$ (ACQ-ISS)
- $\mathit{dom}([G.\mathtt{W}_{\mathtt{strong}}]\,; G.\mathsf{po}\,; [w]) \subseteq I$ (W-STRONG-ISS)

The PPO-ISS condition extends the condition from Def. 6.3. The FWBOB-COV condition arises from Promise's restrictions on promises: a release write cannot be executed if the thread has an unfulfilled promise to the same location, and a release fence cannot be executed if the thread has any unfulfilled promise. Accordingly, we require that when w is issued $G.\mathsf{po}$-previous release writes to the same location and release fences have already been covered. Note that we actually require this from all $G.\mathsf{po}$-previous fences (rather than just release ones). This is not dictated by Promise, but simplifies our proofs. Thus, our proof implies that compilation from Promise to IMM remains correct even if acquire fences "block" promises as release ones. The other conditions in Def. 7.3 are forced by Promise's certification, as demonstrated by the following examples.

Example 7.4. Consider the program and its execution graph on Fig. 8. To certify a promise of a message that corresponds to e_{23}, we need to be able to read the value 2 for x in e_{22} (as e_{23} depends on this value). Thus, the message that corresponds to e_{11} has to be in memory already, *i.e.,* the event e_{11} has to be already issued. This justifies the $G.\mathtt{rfe}\,; G.\mathsf{ppo}$ part of PPO-ISS. The justification for the $G.\mathtt{detour}\,; G.\mathsf{ppo}$ part of PPO-ISS is related to the requirement of certification *for every future memory*. Indeed, in the same example, it is also required that e_{21} was issued before e_{23}: We know that e_{23} is issued after e_{11}, and thus, there is a message of the form $\langle x : 2@(f_{e_{11}}, t_{e_{11}}], \_\rangle$ in

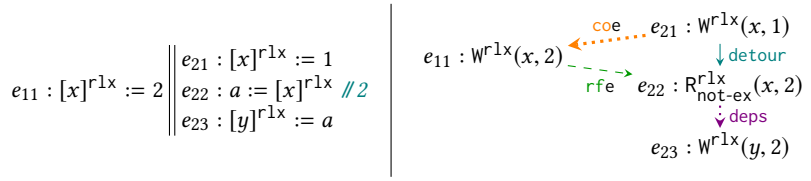


Fig. 8. Demonstration of the necessity of PPO-ISS in the definition of Issuable.

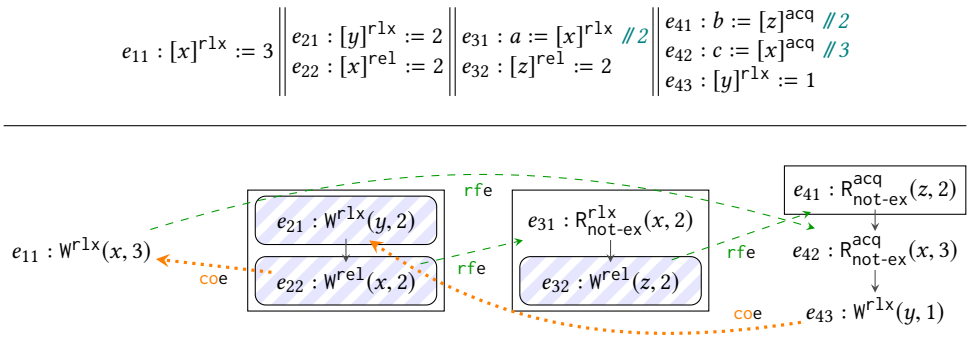


Fig. 9. Demonstration of the necessity of ACQ-ISS in the definition of Issuable. The covered events are marked by ☐ and the issued ones by ◯.

the memory. Had e_{21} not been issued before, the instruction e_{21} would have to add a message of the form $\langle x : 1@(f_{e_{21}}, t_{e_{21}}], \_\rangle$ to the memory during certification. Because e_{22} has to read from $\langle x : 2@(f_{e_{11}}, t_{e_{11}}], \_\rangle$, the timestamp $t_{e_{21}}$ has to be smaller than $t_{e_{11}}$. However, an arbitrary future memory might not have free timestamps in $(0, f_{e_{11}}]$. □

Example 7.5. Consider the program and its execution graph on Fig. 9. Why does e_{43} have to be issued after e_{11}, *i.e.,* why to respect a path $[e_{11}]$; $G.\mathtt{rfe}$; $[G.\mathsf{R}^{\mathsf{acq}}]$; $G.\mathsf{po}$; $[e_{43}]$? In the corresponding state of simulation, the Promise memory has messages related to the issued set with timestamps respecting $G.\mathsf{co}$. Without loss of generality, suppose that the memory contains the messages $\langle y : 2@(1, 2], [y@2]\rangle$, $\langle x : 2@(1, 2], [x@2, y@2]\rangle$, and $\langle z : 2@(1, 2], [x@2, z@2]\rangle$ related to e_{21}, e_{22}, and e_{32} respectively. Since the event e_{41} is covered, the fourth thread has already executed the instruction e_{41}, which is an acquire read. Thus, its current view is updated to include $[x@2, z@2]$. Suppose that e_{43} is issued. Then, the Promise machine has to be able to promise a message $\langle y : 1@(\_, t_{e_{43}}], [y@t_{e_{43}}]\rangle$ for some $t_{e_{43}}$. The timestamp $t_{e_{43}}$ has to be less than 2, which is the timestamp of the message related to e_{21}, since $\langle e_{43}, e_{21}\rangle \in G.\mathsf{co}$. Now, consider a certification run of the fourth thread. In the first step of the run, the thread executes the instruction e_{42}. It is forced to read from $\langle x : 2@(1, 2], [x@2, y@2]\rangle$ since thread's view is equal to $[x@2, z@2]$. Because e_{42} is an acquire read, the thread's current view incorporates the message's view and becomes $[x@2, y@2, z@2]$. After that, the thread cannot fulfill the promise to the location y with the timestamp $t_{e_{43}} < 2$. □

Example 7.6. To see why we need W-STRONG-ISS, revisit the program in Example 3.10. Suppose that we allow to issue $\mathsf{W}^{\mathsf{rlx}}(y, 1)$ before issuing $\mathsf{W}^{\mathsf{rel}}_{\mathsf{strong}}(x, 1)$. Correspondingly, in Promise, the second thread promises a message $\langle y : 1@(1, 2], [y@2]\rangle$ and has to certify it in any future memory. Consider a future memory that contains two messages to location x: an initial one, $\langle x : 0@(0, 0], \bot\rangle$, and

$\langle x : 1@(0,1], [x@1]\rangle$. In this state $c := \mathbf{FADD}^{\mathtt{rlx},\mathtt{rel}}_{\mathtt{strong}}(x,1)$ has to read from the non-initial message and assign 1 to c, since RMWs are required to add messages adjacent to the ones they reads from. After that, $[y]^{\mathtt{rlx}} := c + 1$ is no longer able to fulfill the promise with value 1. □

The full definition of coverable event adds (w.r.t. Def. 6.4) cases related to fence events: for an SC fence to be coverable, all $G.\mathtt{sc}$-previous fence events have to be already covered.

Definition 7.7. An event e is called *coverable* in G and $\langle C, I\rangle$, denoted $e \in \mathsf{Coverable}(G, C, I)$, if $e \in G.\mathsf{E}$, $dom(G.\mathtt{po}\,;[e]) \subseteq C$, and either (i) $e \in G.\mathtt{W} \cap I$; (ii) $e \in G.\mathtt{R}$ and $dom(G.\mathtt{rf}\,;[e]) \subseteq I$; (iii) $e \in G.\mathsf{F}^{\sqsubset \mathtt{sc}}$; or (iv) $e \in G.\mathsf{F}^{\mathtt{sc}}$ and $dom(G.\mathtt{sc}\,;[e]) \subseteq C$.

By further requiring that traversals configurations $\langle C, I\rangle$ of an execution G satisfy $I \cap G.\mathtt{W}^{\mathtt{rel}} \subseteq C$ and $codom([C]\,;G.\mathtt{rmw}) \subseteq C$, Prop. 6.5 is extended to the updated definition of the traversal strategy.

7.3 Thread Step Simulation

Next, we refine the simulation relation from §6.3. The relation $\mathcal{I}_i(G, TC, \langle TS, \mathcal{S}, M\rangle, F, T)$ has an additional parameter $F : I \to \mathbb{Q}$, which is used to assign lower bounds of a timestamp interval to issued writes (T assigns upper bounds). We define this relation to hold if the following conditions are met (for conciseness we omit the "G." prefix):[12]

(1) F and T agree with $\mathtt{co}$ and reflect the requirements on timestamp intervals:
- $\forall w \in \mathsf{E} \cap \mathsf{Init}.\ T(w) = F(w) = 0$ and $\forall w \in I \setminus \mathsf{Init}.\ F(w) < T(w)$
- $\forall \langle w, w'\rangle \in [I]\,;\mathtt{co}\,;[I].\ T(w) \le F(w')$ and $\forall \langle w, w'\rangle \in [I]\,;\mathtt{rf}\,;\mathtt{rmw}\,;[I].\ T(w) = F(w')$

(2) Non-initialization messages in M have counterparts in I:
- $\forall \langle x : \_@(f,t], \_\rangle \in M.\ t \neq 0 \Rightarrow \exists w \in I.\ \mathtt{loc}(w) = x \wedge F(w) = f \wedge T(w) = t$
- $\forall \langle w, w'\rangle \in [I]\,;\mathtt{co}\,;[I].\ T(w) = F(w') \Rightarrow \langle w, w'\rangle \in \mathtt{rf}\,;\mathtt{rmw}$

(3) The SC view $\mathcal{S}$ corresponds to write events that are "before" covered SC fences:
- $\mathcal{S} = \lambda x.\ \max T[\mathtt{W}(x) \cap dom(\mathtt{rf}^?\,;\mathtt{hb}\,;[C \cap \mathsf{F}^{\mathtt{sc}}])]$

(4) Issued events have corresponding messages in memory:
- $\forall w \in I.\ \langle \mathtt{loc}(w) : \mathtt{val}(w)@(F(w), T(w)], \mathrm{view}(T, w)\rangle \in M$, where:
  - $\mathrm{view}(T, w) \triangleq (\lambda x.\ \max T[\mathtt{W}(x) \cap dom(\mathtt{vf}\,;\mathtt{release}\,;[w])]) \sqcup [\mathtt{loc}(w)@T(w)]$
  - $\mathtt{vf} \triangleq \mathtt{rf}^?\,;(\mathtt{hb}\,;[\mathsf{F}^{\mathtt{sc}}])^?\,;\mathtt{sc}^?\,;\mathtt{hb}^?$

(5) For every promise, there exists a corresponding issued uncovered event w:
- $\forall \langle x : v@(f,t], view\rangle \in P.\ \exists w \in \mathsf{E}_i \cap I \setminus C.$
  $\mathtt{loc}(w) = x \wedge \mathtt{val}(w) = v \wedge F(w) = f \wedge T(w) = t \wedge view = \mathrm{view}(T, w)$

(6) Every issued uncovered event w of thread i has a corresponding promise in P. Its message view includes the singleton view $[\mathtt{loc}(w)@T(w)]$ and the thread's release view rel (third component of $\mathcal{V}$). If w is an RMW write, and its read part is reading from an issued write p, the view of the message that corresponds to p is also included in w's message view.
- $\forall w \in \mathsf{E}_i \cap I \setminus (C \cup codom([I]\,;\mathtt{rf}\,;\mathtt{rmw})).$
  $\langle \mathtt{loc}(w) : \mathtt{val}(w)@(F(w), T(w)], [\mathtt{loc}(w)@T(w)] \sqcup \mathrm{rel}(x)\rangle \in P$
- $\forall w \in \mathsf{E}_i \cap I \setminus C, p \in I.\ \langle p, w\rangle \in \mathtt{rf}\,;\mathtt{rmw} \Rightarrow$
  $\langle \mathtt{loc}(w) : \mathtt{val}(w)@(F(w), T(w)], [\mathtt{loc}(w)@T(w)] \sqcup \mathrm{rel}(x) \sqcup \mathrm{view}(T, p)\rangle \in P$

(7) The three components $\langle \mathrm{cur}, \mathrm{acq}, \mathrm{rel}\rangle$ of $\mathcal{V}$ are justified by graph paths:
- $\mathrm{cur} = \lambda x.\ \max T[\mathtt{W}(x) \cap dom(\mathtt{vf}\,;[\mathsf{E}_i \cap C])]$
- $\mathrm{acq} = \lambda x.\ \max T[\mathtt{W}(x) \cap dom(\mathtt{vf}\,;(\mathtt{release}\,;\mathtt{rf})^?\,;[\mathsf{E}_i \cap C])]$
- $\mathrm{rel} = \lambda x, y.\ \max T[\mathtt{W}(x) \cap (dom(\mathtt{vf}\,;[(\mathtt{W}^{\mathtt{rel}}(y) \cup \mathsf{F}^{\sqsupset \mathtt{rel}}) \cap \mathsf{E}_i \cap C]) \cup \mathtt{W}(y) \cap \mathsf{E}_i \cap C)]$

[12] To relate the timestamps in the different views to relations in G (items (3),(4),(7)), we use essentially the same definitions that were introduced by Kang et al. [2017] when they related the promise-free fragment of Promise to a declarative model.

(8) The thread local state σ matches the covered events ($\sigma.\mathtt{G.E} = C \cap \mathtt{E}_i$), and can always reach the execution graph G ($\exists \sigma'.\ \sigma \to_i^* \sigma' \wedge \sigma'.\mathtt{G} = G|_i$).

We also state a version of Prop. 6.6 for the new relation.

Proposition 7.8. *If $\mathcal{I}_i(G, TC, \langle TS, \mathcal{S}, M\rangle, F, T)$ and $G \vdash TC \to_i TC'$ hold, then there exist TS', $\mathcal{S}'$, M', F', T' such that $\langle TS, \mathcal{S}, M\rangle \to^+ \langle TS', \mathcal{S}', M'\rangle$ and $\mathcal{I}_i(G, TC', \langle TS', \mathcal{S}', M'\rangle, F', T')$ hold.*

7.4 Certification

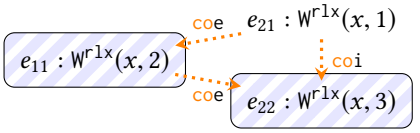
We move on to the construction of certification graphs. First, the set of events of G^{crt} is extended:

$$G^{\text{crt}}.\mathtt{E} \triangleq C \cup I \cup \mathit{dom}(G.\mathtt{po}\,;\,[I \cap G.\mathtt{E}_i]) \cup (\mathit{dom}(G.\mathtt{rmw}\,;\,[I \cap G.\mathtt{E}_{\neq i}]) \setminus \mathit{codom}([G.\mathtt{E} \setminus \mathit{codom}(G.\mathtt{rmw})]\,;\,G.\mathtt{rfi}))$$

It additionally contains read parts of issued RMWs in other threads (excluding those reading locally from a non-RMW write). They are needed to preserve release sequences to issued writes in G^{crt}.

The rmw, sc and dependencies components of G^{crt} are the same as in (restricted) G ($G^{\text{crt}}.\mathtt{x} = [G^{\text{crt}}.\mathtt{E}]\,;G.\mathtt{x}\,;[G^{\text{crt}}.\mathtt{E}]$ for $\mathtt{x} \in \{\mathtt{rmw}, \mathtt{addr}, \mathtt{data}, \mathtt{ctrl}, \mathtt{casdep}, \mathtt{sc}\}$) as in §6.4. However, $G.\mathtt{co}$ edges have to be altered due to the future memory quantification in Promise certifications.

Example 7.9. Consider the annotated execution G and its traversal configuration ($C = \emptyset$ and $I = \{e_{11}, e_{22}\}$) shown in the inlined figure. Suppose that $\mathcal{I}_{i_2}(G, \langle C, I\rangle, \langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle, F, T)$ holds for some σ, $\mathcal{V}$, P, M, $\mathcal{S}$, F and T. Hence, there are messages of the form $\langle x : 2@(F(e_{11}), T(e_{11})], \_\rangle$ and $\langle x : 3@(F(e_{22}), T(e_{22})], \_\rangle\}$ in M and $F(e_{11}) < T(e_{11}) \le F(e_{22}) < T(e_{22})$.



During certification, we have to execute the instruction related to e_{21} and add a corresponding message to M. Since certification is required for every future memory $M_{\text{fut}} \supseteq M$, it might be the case that here is no free timestamp t' in M_{fut} such that $t' \le F(e_{11})$. Thus, our chosen timestamps cannot agree with $G.\mathtt{co}$. However, if we place e_{21} as the immediate predecessor of e_{22} in $G^{\text{crt}}.\mathtt{co}$, we may use the splitting feature of Promise: the promised message $\langle x : 3@(F(e_{22}), T(e_{22})], \_\rangle\}$ can be split into two messages $\langle x : 1@(F(e_{22}), t], \_\rangle\}$ and $\langle x : 3@(t, T(e_{22})], \_\rangle\}$ for any t such that $F(e_{22}) < t < T(e_{22})$. To do so, we need the non-issued writes of the certified thread to be immediate predecessors of the issued ones in $G^{\text{crt}}.\mathtt{co}$. By performing such split, we do not "allocate" new timestamp intervals, which allows us to handle arbitrary future memories. Note that if we had writes to other locations to perform during the certification, with no possible promises to split, we would need them to be placed last in $G^{\text{crt}}.\mathtt{co}$, so we can relate them to messages whose timestamps are larger than all timestamps in M_{fut}. □

Following Example 7.9, we define $G^{\text{crt}}.\mathtt{co}$ to consist of all pairs $\langle w, w'\rangle$ such that $w, w' \in G^{\text{crt}}.\mathtt{E} \cap G.\mathtt{W}$, $G.\mathtt{loc}(w) = G.\mathtt{loc}(w')$, and either $\langle w, w'\rangle \in ([I]\,;G.\mathtt{co}\,;[I] \cup [I]\,;G.\mathtt{co}\,;[G^{\text{crt}}.\mathtt{E}_i] \cup [G^{\text{crt}}.\mathtt{E}_i]\,;G.\mathtt{co}\,;[G^{\text{crt}}.\mathtt{E}_i])^+$, or there is no such path, $w \in I$, and $w' \in G^{\text{crt}}.\mathtt{E}_i \setminus I$. This construction essentially "pushes" the non-issued writes of the certified thread to be as late as possible in $G^{\text{crt}}.\mathtt{co}$.

The definition of $G^{\text{crt}}.\mathtt{rf}$ is also adjusted to be in accordance with $G^{\text{crt}}.\mathtt{co}$:

$$G^{\text{crt}}.\mathtt{rf} \triangleq G.\mathtt{rf}\,;[D] \cup \bigcup\nolimits_{x \in \mathsf{Loc}}([G.\mathtt{W}(x)]\,;G.\mathtt{bvf}\,;[G.\mathtt{R}(x) \cap G^{\text{crt}}.\mathtt{E} \setminus D] \setminus G^{\text{crt}}.\mathtt{co}\,;G.\mathtt{bvf})$$
where $D = G^{\text{crt}}.\mathtt{E} \cap (C \cup I \cup G.\mathtt{E}_{\neq i} \cup \mathit{dom}(G.\mathtt{rfi}^?\,;G.\mathtt{ppo}\,;[I]) \cup \mathit{codom}(G.\mathtt{rfe}\,;[G.\mathtt{R}^{\mathtt{acq}}]))$ and
$G.\mathtt{bvf} = (G.\mathtt{rf}\,;[D])^?\,;(G.\mathtt{hb}\,;[G.\mathtt{F}^{\mathtt{sc}}])^?\,;G.\mathtt{sc}^?\,;G.\mathtt{hb}$

The set of determined events is extended to include acquire read events which read externally, *i.e.,* the ones potentially engaged in synchronization.

For the certification graph G^{crt} presented here, we prove a version of Prop. 6.7, *i.e.,* show that the graph is IMM-consistent and TC^{crt} is its traversal configuration, and adapt Prop. 6.8 as follows.

Proposition 7.10. *Suppose that* $\mathcal{I}_i(G, TC, \langle TS, \mathcal{S}, M\rangle, F, T)$ *holds. Then, for every* $M_{fut} \supseteq M$ *and* $\mathcal{S}_{fut} \geq \mathcal{S}$, $\mathcal{I}_i^{\mathrm{crt}}(G^{\mathrm{crt}}, TC^{\mathrm{crt}}, \langle TS, \mathcal{S}_{fut}, M_{fut}\rangle, F, T)$ *holds.*

Here, $\mathcal{I}_i^{\mathrm{crt}}$ is a modified simulation relation, which differs to $\mathcal{I}_i$ in the following parts:

(2) Since certification begins from an arbitrary future memory, we cannot require that *all* messages in memory have counterparts in I. Here, it suffices to assert that all RMW writes are issued ($codom(G^{\mathrm{crt}}.\mathtt{rmw}) \subseteq I$), and for every non-issued write either it is last in $G^{\mathrm{crt}}.\mathtt{co}$ or its immediate successor is in the same thread ($[G^{\mathrm{crt}}.\mathtt{E} \setminus I]\,;\, G^{\mathrm{crt}}.\mathtt{co}|_{\mathrm{imm}} \subseteq G^{\mathrm{crt}}.\mathtt{po}$). The latter allows us to split existing messages to obtain timestamp intervals for non-issued writes during certification (see Example 7.9).

(3) Since certification begins from arbitrary future SC view, $\mathcal{S}$ may not correspond to G^{crt}. Nevertheless, SC fences cannot be executed in the certification run, and we can simply require that all SC fences are covered ($G^{\mathrm{crt}}.\mathtt{F^{sc}} \subseteq C^{\mathrm{crt}}$).

We also show that a version of Prop. 7.8 holds for $\mathcal{I}^{\mathrm{crt}}$. It allows us to prove a strengthened version Prop. 7.8, which also concludes that new Promise thread state is certifiable, in a similar way we prove Prop. 6.9.

Proposition 7.11. *If* $\mathcal{I}_i(G, TC, \langle TS, \mathcal{S}, M\rangle, F, T)$ *and* $G \vdash TC \rightarrow_i TC'$ *hold, then there exist* $TS', \mathcal{S}', M', F', T'$ *such that* $\langle TS, \mathcal{S}, M\rangle \rightarrow^+ \langle TS', \mathcal{S}', M'\rangle$ *and* $\mathcal{I}_i(G, TC', \langle TS', \mathcal{S}', M'\rangle, F', T')$ *hold, and for every* $\mathcal{S}_{fut} \geq \mathcal{S}', M_{fut} \supseteq M'$, *there exist* $TS'', \mathcal{S}'_{fut}, M'_{fut}$ *such that* $\langle TS', \mathcal{S}_{fut}, M_{fut}\rangle \rightarrow^* \langle TS'', \mathcal{S}'_{fut}, M'_{fut}\rangle$ *and* $TS''.\mathtt{prm} = \emptyset$.

8 RELATED WORK

Together with the introduction of the promising semantics, Kang et al. [2017] provided a declarative presentation of the promise-free fragment of the promising model. They established the adequacy of this presentation using a simulation relation, which resembles the simulation relation that we use in §7. Nevertheless, since their declarative model captures only the promise-free fragment of Promise, the simulation argument is much simpler, and no certification condition is required. In particular, their analogue to our traversal strategy would simply cover the events of the execution graph following $\mathtt{po} \cup \mathtt{rf}$.

To establish the correctness of compilation of the promising semantics to POWER, Kang et al. [2017] followed the approach of Lahav and Vafeiadis [2016]. This approach reduces compilation correctness to POWER to (*i*) the correctness of compilation to the POWER model strengthened with $\mathtt{po} \cup \mathtt{rf}$ acyclicity; and (*ii*) the soundness of local reorderings of memory accesses. To establish (*i*), Kang et al. [2017] wrongly argued that the strengthened POWER-consistency of mapped promise-free execution graphs imply the promise-free consistency of the source execution graphs. This is not the case due to SC fences, which have relatively strong semantics in the promise-free declarative model (see [Podkopaev et al. 2018, Appendix D] for a counter example). Nevertheless, our proof shows that the compilation claim of Kang et al. [2017] is correct. We note also that, due to the limitations of this approach, Kang et al. [2017] only claimed the correctness of a less efficient compilation scheme to POWER that requires `lwsync` barriers after acquire loads rather than (cheaper) control dependent `isync` barriers. Finally, this approach cannot work for ARM as it relies on the relative strength of POWER's preserved program order.

Podkopaev et al. [2017] proved (by paper-and-pencil) the correctness of compilation from the promising semantics to ARMv8. Their result handled only a restricted subset of the concurrency features of the promising semantics, leaving release/acquire accesses, RMWs, and SC fences out of

scope. In addition, as a model of ARMv8, they used an operational model, ARMv8-POP [Flur et al. 2016], that was later abandoned by ARM in favor of a stronger different declarative model [Pulte et al. 2018]. Our proof in this paper is mechanized, supports all features of the promising semantics, and uses the recent declarative model of ARMv8.

Wickerson et al. [2017] developed a tool, based on the Alloy solver, that can be used to test the correctness of compiler mappings. Given the source and target models and the intended compiler mapping, their tool searches for minimal litmus tests that witness a bug in the mapping. While their work concerns automatic bug detection, the current work is focused around formal verification of the intended mappings. In addition, their tool is limited to declarative specifications, and cannot be used to test the correctness of the compilation of the promising semantics.

Finally, we note that IMM is weaker than the ARMv8 memory model of Pulte et al. [2018]. In particular, IMM is not multi-copy atomic (see Example 3.8); its release writes provide weaker guarantees (allowing in particular the so-called 2+2W weak behavior [Lahav et al. 2016; Maranget et al. 2012]); it does not preserve address dependencies between reads (allowing in particular the "big detour" weak behavior [Pulte et al. 2018]); and it allows "write subsumption" [Flur et al. 2016; Pulte et al. 2018]. Formally, this is a result of not including `fr` and `co` in a global acyclicity condition, but rather having them in a C/C++11-like coherence condition. While Pulte et al. [2018] consider these strengthenings of the ARMv8 model as beneficial for its simplicity, we do not see IMM as being much more complicated than the ARMv8 declarative model. (In particular, IMM's derived relations are not mutually recursive.) Whether or not these weaknesses of IMM in comparison to ARMv8 allow more optimizations and better performance is left for future work.

9 CONCLUDING REMARKS

We introduced a novel intermediate model, called IMM, as a way to bridge the gap between language-level and hardware models and modularize compilation correctness proofs. On the hardware side, we provided (machine-verified) mappings from IMM to the main multi-core architectures, establishing IMM as a common denominator of existing hardware weak memory models. On the programming language side, we proved the correctness of compilation from the promising semantics, as well as from a fragment of (R)C11, to IMM.

In the future, we plan to extend our proof for verifying the mappings from full (R)C11 to IMM as well as to handle infinite executions with a more expressive notion of a program outcome. We believe that IMM can be also used to verify the implementability of other language-level models mentioned in §1. This might require some modifications of IMM (in the case it is too weak for certain models) but these modifications should be easier to implement and check over the existing mechanized proofs. Similarly, new (and revised) hardware models could be related to (again, a possibly modified version of) IMM. Specifically, it would be nice to extend IMM to support mixed-size accesses [Flur et al. 2017] and hardware transactional primitives [Chong et al. 2018; Dongol et al. 2017]. On a larger scope, we believe that IMM may provide a basis for extending CompCert [Leroy 2009; Ševčík et al. 2013] to support modern multi-core architectures beyond x86-TSO.

ACKNOWLEDGMENTS

We thank Orestis Melkonian for his help with Coq proof concerning the POWER model in the context of another project, and the POPL'19 reviewers for their helpful feedback. The first author was supported by RFBR (grant number 18-01-00380). The second author was supported by the Israel Science Foundation (grant number 5166651), and by Len Blavatnik and the Blavatnik Family foundation.

REFERENCES

Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *ASPLOS 2018*. ACM, New York, 405–418. https://doi.org/10.1145/3173162.3177156

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *ESOP 2015 (LNCS)*, Vol. 9032. Springer, Berlin, Heidelberg, 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and Compiling C/C++ Concurrency: From C++11 to POWER. In *POPL 2012*. ACM, New York, 509–520. https://doi.org/10.1145/2103656.2103717

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *POPL 2011*. ACM, New York, 55–66. https://doi.org/10.1145/1925844.1926394

Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *MSPC 2014*. ACM, New York, Article 7, 6 pages. https://doi.org/10.1145/2618128.2618134

Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the concurrency semantics of an LLVM fragment. In *CGO 2017*. IEEE Press, Piscataway, NJ, USA, 100–110. https://doi.org/10.1109/CGO.2017.7863732

Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding Thin-Air Reads with Event Structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 70:1–70:27. https://doi.org/10.1145/3290383

Nathan Chong, Tyler Sorensen, and John Wickerson. 2018. The Semantics of Transactions and Weak Memory in x86, Power, ARM, and C++. In *PLDI 2018*. ACM, New York, 211–225. https://doi.org/10.1145/3192366.3192373

Will Deacon. 2017. The ARMv8 Application Level Memory Model. Retrieved June 27, 2018 from https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time. In *PLDI 2018*. ACM, New York, 242–255. https://doi.org/10.1145/3192366.3192421

Brijesh Dongol, Radha Jagadeesan, and James Riely. 2017. Transactions in Relaxed Memory Architectures. *Proc. ACM Program. Lang.* 2, POPL, Article 18 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158106

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *POPL 2016*. ACM, New York, 608–621. https://doi.org/10.1145/2837614.2837615

Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size Concurrency: ARM, POWER, C/C++11, and SC. In *POPL 2017*. ACM, New York, 429–442. https://doi.org/10.1145/3009837.3009839

Alan Jeffrey and James Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *LICS 2016*. ACM, New York, 759–767. https://doi.org/10.1145/2933575.2934536

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *POPL 2017*. ACM, New York, 175–189. https://doi.org/10.1145/3009837.3009850

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-acquire Consistency. In *POPL 2016*. ACM, New York, 649–662. https://doi.org/10.1145/2837614.2837643

Ori Lahav and Viktor Vafeiadis. 2016. Explaining Relaxed Memory Models with Program Transformations. In *FM 2016*. Springer, Cham, 479–495. https://doi.org/10.1007/978-3-319-48989-6_29

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

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2016. Counterexamples and Proof Loophole for the C/C++ to POWER and ARMv7 Trailing-Sync Compiler Mappings. *CoRR* abs/1611.01507 (2016). http://arxiv.org/abs/1611.01507

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *POPL 2005*. ACM, New York, 378–391. https://doi.org/10.1145/1040305.1040336

Mapping 2016. C/C++11 mappings to processors. Retrieved June 27, 2018 from http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf.

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *TPHOLs 2009 (LNCS)*, Vol. 5674. Springer, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics that Permits Optimisation and Avoids Thin-Air Executions. In *POPL 2016*. ACM, New York, 622–633. https://doi.org/10.1145/2837614.2837616

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2017. Promising Compilation to ARMv8 POP. In *ECOOP 2017 (LIPIcs)*, Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:28. https://doi.org/10.4230/LIPIcs.ECOOP.2017.22

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2018. Coq proof scripts and supplementary material for this paper, available at http://plv.mpi-sws.org/imm/.

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

RISC-V 2018. The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA. Available at https://github.com/riscv/riscv-isa-manual/releases/download/draft-20180731-e264b74/riscv-spec.pdf [Online; accessed 23-August-2018].

RISCV in herd 2018. RISCV: herd vs. operational models. Retrieved October 22, 2018 from http://diy.inria.fr/cats7/riscv/

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *POPL 2015*. ACM, New York, 209–220. https://doi.org/10.1145/2676726.2676995

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22. https://doi.org/10.1145/2487241.2487248

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *POPL 2017*. ACM, New York, 190–204. https://doi.org/10.1145/3009837.3009838

Sizhuo Zhang, Muralidaran Vijayaraghavan, Andrew Wright, Mehdi Alipour, and Arvind. 2018. Constructing a Weak Memory Model. In *ISCA 2018*. IEEE Computer Society, Washington, DC, 124–137. https://doi.org/10.1109/ISCA.2018.00021