Correct Translation between Weak Memory Model Architectures

Sprokholt, D.G.

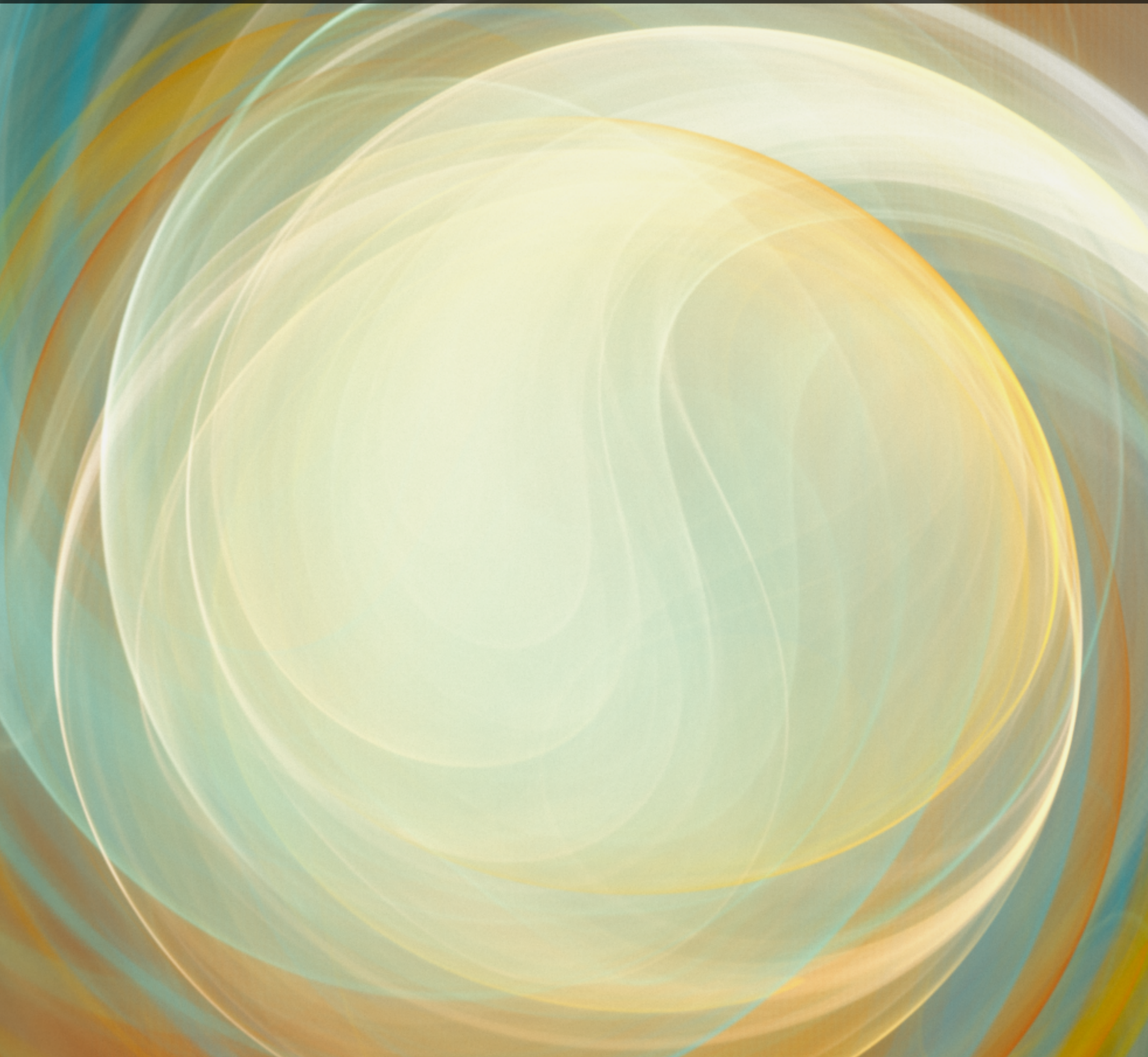
DOI
10.4233/uuid:8dc4a658-85bf-4243-993c-2bb511abe5f1

Publication date
2025

Document Version
Final published version

Citation (APA)
Sprokholt, D. G. (2025). *Correct Translation between Weak Memory Model Architectures*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:8dc4a658-85bf-4243-993c-2bb511abe5f1

Correct Translation between Weak Memory Model Architectures



Dennis Sprokholt

Correct Translation between Weak Memory Model Architectures

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus, prof. dr. ir. T.H.J.J. van der Hagen;
Chair of the Board for Doctorates
to be defended publicly on
Friday 12 December 2025 at 10:00 o'clock

by

Dennis Guido SPROKHOLT

Master of Science in Computer Science,
Utrecht University, the Netherlands

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus	chairperson
Prof.dr. K.G. Langendoen	Delft University of Technology, *promotor*
Dr. S.S. Chakraborty	Delft University of Technology, *copromotor*

Independent members:

Prof.dr.ir. G.N. Gaydadjiev	Delft University of Technology
Prof.dr. P.A. Abdulla	Uppsala University, Sweden
Prof.dr. B. Dongol	University of Surrey, United Kingdom
Prof.dr. S.B. Scholz	Radboud University
Prof.dr. M. Huisman	University of Twente
Prof.dr. M.M. de Weerdt	Delft University of Technology, *reserve member*

Prof.dr. Eelco Visser (Delft University of Technology) was the original promotor of this research until his untimely passing on April 5th, 2022.

The work in this dissertation has been carried out at the Delft University of Technology, under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Printed by: Gildeprint – `https://gildeprint.nl`

IPA Dissertation Series: 2025-16

ISBN 978-94-6518-172-1

An electronic version of this dissertation is available at `https://repository.tudelft.nl`

Acknowledgements

Thank you to my supervisor, Soham, for your guidance, encouragement, and technical expertise, which were essential to produce the contents of this dissertation. You taught me that finding collaborators to help solve research problems is equally important to finding the problems themselves; this led to many collaborations with diverse perspectives on interesting problems. Thank you to my promotor, Koen, for your practical perspective, insightful questions, and for encouraging me to look at the bigger picture to shape the coherent narrative of this dissertation. Your guarding of the process—particularly by insisting on deadlines—also helped finish it *on time*. Thank you to my late promotor, Eelco, for founding the Programming Languages group and for believing in me enough to offer me a PhD position in it.

Thank you to my collaborators Rodrigo, Redha, Pramod, Tom, Martin, Sebastian, Theo, Krishna, Anish, and Yifan. Our frequent discussions and differing points of view were essential for iterating on the ideas that led to our co-authored papers. Thank you to my committee members, Parosh Aziz Abdulla, Brijesh Dongol, Sven-Bodo Scholz, Marieke Huisman, and Georgi Gaydadjiev, for your time, effort, and valuable feedback on this dissertation. Thank you to the countless anonymous conference reviewers for providing feedback on the included papers.

Thank you to the members of the Delft Programming Languages group for the regular coffee breaks, where we could share our suffering. Thank you (in random order) Luka for “singing” together, Xulei for the plants, Cas for sitting in the hallway eggs, Lucas for the monitor box, Kobe for the whiteboard scribbles, Bohdan for “buying my car”, Jaro for the pure FP chats, and Max for being in my gym; thank you Jeff, Andreea, Aron, Danny, Jasper, Casper, Peter, Alex, Elmer, Jesper, Gabriël, Arjen, Sára, and Daniel. Thank you to the interns: Arthur for taking my pictures, Thomas for teaching me to climb, Ewen, Micol, and Houda. Thank you to the support staff Roniet, Shelly, Sophie, Myrthe, and Marja. Thank you to my students, Pieter, Kuba, and Einar; working with you helped me recognize the boundaries of my own understanding — both technical and non-technical. Thank you to all the other people I have enjoyed spending time with while visiting conferences and summer schools.

Thank you to the people who made these PhD years more enjoyable outside university walls. Thank you for the Brouwhuis visits and beach rituals, Máté, Jeffrey, and David. Thank you, Kevin and Joris, for playing Among Us—especially during lockdown. Thank you, Qiao and Labrinus, for the boat trip. Thank you, Thomas, Cas, Max, Kobe, Luka, Jaro, and David, for climbing (and falling) with me.

Thank you to my father and mother, for always supporting and believing in me. Thank you to my sister Laura, and Pieter. Finally, thank you to my girlfriend Dān (丹) for your encouragement, support, and patience throughout this journey.

English Summary

This dissertation is about translating concurrent programs between computer architectures. Legacy programs—built-for and tested-on x86—behave differently on newer architectures, such as Arm and RISC-V. Particularly, *weak memory* behaviors emerge when two micro-architectural features interact: *(i)* concurrency, where multiple CPU cores simultaneously execute parts of a program, and *(ii)* out-of-order execution, where a CPU core reorders instructions to increase throughput. Programs can *non-deterministically* show one of various weak memory behaviors, meaning it could behave differently when executing again. Those behaviors differ between architectures. When migrating programs from x86 to Arm or RISC-V, the same program could non-deterministically show behaviors never observed on x86.

In part I, we look at *binary translators*, which are software systems that translate compiled binary programs between architectures. We study the translation process of three such real-world systems, identify errors in their translation of concurrency primitives, and fix them. We propose mathematically-rigorous weak memory models for these translators. We then define mapping schemes to translate concurrency primitives one-by-one from x86 to Arm and RISC-V. With the formal semantics, we prove those mapping schemes correct in the Agda proof assistant.

In part II, we study the common structure of our weak memory proofs written in Agda. As those proofs are often large, complex, and rigid, we identify their common structures for which we identify domain-specific abstractions. We implement those abstractions in our novel Agda proof framework BURROW to greatly simplify writing future weak memory proofs.

In part III, we use *dynamic analysis* to identify weak behaviors that were never seen on x86 but could appear on Arm. Our analysis *simulates* the program's execution with the formal weak memory semantics of x86 and Arm. This analysis identifies only the new behaviors the program shows in practice. After finding any new behavior on Arm, we judiciously modify the program to eliminate only that behavior.

Nederlandse Samenvatting

Deze dissertatie betreft het vertalen van gelijktijdige programma's tussen verschillende computerarchitecturen. Computerprogramma's die zijn geschreven voor en getest op x86 gedragen zich anders op nieuwe architecturen, zoals Arm en RISC-V. Vooral *weak memory* gedrag verschilt, wat zichtbaar is wanneer twee eigenschappen van microarchitecturen samen komen: *(i)* gelijktijdigheid, waar meerdere processesoren tegelijkertijd delen van hetzelfde programma uitvoeren, en *(ii) out-of-order* programma-uitvoering, waar de processor instructies in een andere volgorde uitvoert dan zoals ze in het programma staan. Een programma kan willekeurig één van meerdere *weak memory* gedragingen laten zien, wat betekent dat het programma zich anders kan gedragen wanneer het opnieuw wordt uitgevoerd. Daarnaast verschilt het mogelijke *weak memory* gedrag per architectuur. Wanneer een programma wordt vertaald van x86 naar Arm of RISC-V, kan het nieuw gedrag laten zien, dat nooit eerder was waargenomen op x86.

Deel I gaat over *binaire vertalers* die binaire programma's vertalen tussen computerarchitecturen. We bestuderen het vertaalproces van drie bestaande vertaalsystemen, waarin we fouten identificeren en repareren in de vertaling van gelijktijdigheidsinstructies. We introduceren rigoreuze wiskundige *weak memory* modellen voor die vertalers. Daarna definiëren we vertaalschema's om gelijktijdigheidsinstructies één voor één te vertalen van x86 naar Arm en RISC-V. Met de formele modellen bewijzen we die vertaalschema's correct in de Agda bewijsassistent.

In deel II bestuderen we de structuur van *weak memory* bewijzen in Agda. Die bewijzen zijn vaak groot, complex, en moeilijk aan te passen. Om die complexiteit te verlagen definiëren we domeinspecifieke abstracties voor hun gemeenschappelijke bewijsstructuur. We implementeren die abstracties in onze Agda bibliotheek BURROW waarmee *weak memory* transformaties makkelijker zijn te bewijzen.

In deel III gebruiken we *dynamische analyse* om *weak memory* gedrag te identificeren dat niet op x86 zichtbaar is maar wel op Arm. Onze analyse simuleert de uitvoering van het programma volgens de formele semantiek van x86 en Arm. Na het vinden van een nieuw gedrag op Arm passen we het programma aan om dat gedrag uit te sluiten. Onze analyse identificeert enkel de gedragingen die in de praktijk op Arm zichtbaar zijn.

Contents

II Proof Mechanization

III Dynamic Analysis

Introduction

Modern computing is concurrent, where multiple CPU cores execute a specific task *together*, while communicating through shared memory. Unfortunately, reasoning about a concurrent program is challenging because it behaves non-deterministically – which means its behavior could differ when executing again. Non-determinism appears when program threads interleave arbitrarily. In addition, when a program executes on computer hardware, more *weak* behaviors may non-deterministically appear that cannot be explained by thread interleaving. Those weak behaviors could occur when instructions execute out-of-order—to increase throughput—or when changes to shared program state do not immediate synchronize between threads.

Additionally, computer architectures differ, causing programs to show *different non-deterministic behaviors* when executing on different architectures. If we naively translate a program between two architectures without considering those weak behaviors, the resulting program could show unforeseen behaviors that never appeared on the original architecture. That difference between architectures thus poses a challenge when translating any program written for one architecture to another, while that translation is crucial to ensure the program's longevity across changes in the hardware landscape. This thesis addresses that challenge by *formally reasoning* about translating *concurrent programs* between *weak memory computer architectures*.

Concurrency on Computer Architectures. The observable weak behaviors vary between architectures. For instance, the predominant x86 (Owens et al., 2009) and Arm (Alglave et al., 2021) architectures show different weak behaviors, as illustrated in Figure 1. Therefore, translating a program from x86 to Arm is prone to errors, as it may unexpectedly introduce additional weak behaviors with new program outputs.

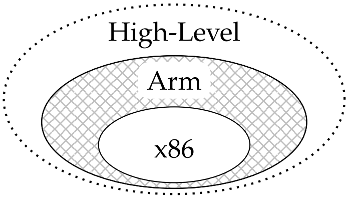


Figure 1: *Problem Illustration.* A concurrent program written in a high-level language may show one of many different behaviors, but *only a subset* of those appears on x86. When recompiling to Arm, unforeseen behaviors –that were not observed on x86– suddenly appear.

When we only have the compiled x86 program, but not the original source code, we do not know which additional behaviors were allowed by that source. However, even when we do have its source code in a high-level programming language, it could *implicitly* rely upon characteristics specific to the computer architecture used during development; for instance, when its developers only considered that one architecture or never tested on others. Upon recompiling to another architecture, such as Arm, that same program may suddenly show weak behaviors and outputs that were not observed before on x86.

Formal Reasoning. Although a program may behave non-deterministically, meaning its behavior differs between executions, those behaviors are *not arbitrary*. Any architecture preserves some order among instructions, described in a *formal semantics* of its *memory consistency model*. Those semantics mathematically describe the order among operations on all program threads, when executed on its corresponding architecture. Any processor implementing that architecture must follow its corresponding memory consistency model. With *formal reasoning* we refer to a *mathematically rigorous* approach to specifying and analyzing those weak memory semantics.

Through rigorous inspection of semantics, we can ensure programs behave as intended in all situations. By extension, analyzing program semantics often helps improve program *performance*. When ignoring semantics, we cannot be certain whether a particular transformation is correct. We might thus cautiously avoid it, hoping not to introduce errors but also miss out on its optimization opportunities. In contrast, when we know those semantics, we can judiciously pick any correct transformation, among which we can select the optimal one.

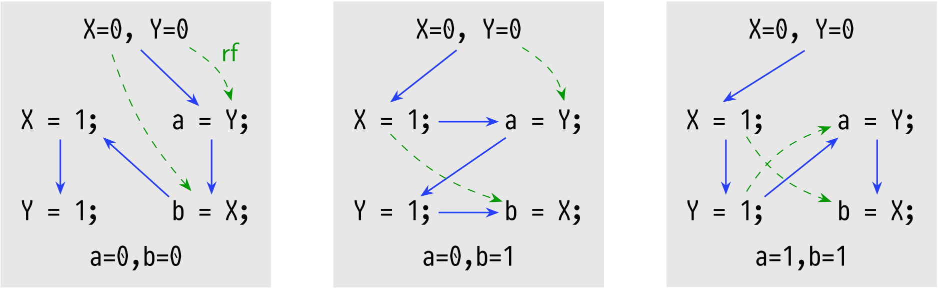
WEAK MEMORY EXAMPLE

To illustrate the challenges of concurrent programming, consider the program below. It has *two* threads, with global variables X and Y, initialized to 0 and shared between the threads, while a and b are local to the right thread. The left thread *passes messages* by writing 1 to X and Y, which the right thread reads in reverse order.

```
   X=0, Y=0
X = 1; || a = Y;
Y = 1; || b = X;
```

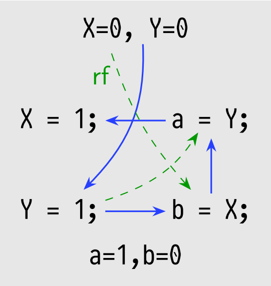
(Message-Passing)

When considering only thread interleaving, corresponding to *sequential consistency* (Lamport, 1979), we perceive a global order between operations that preserves their order within each thread; instructions do no execute out-of-order but threads interleave arbitrarily. For instance, three such orders are shown with solid blue arrows:



Depending on the interleaving at runtime, different assignments to X and Y precede their observation by b and a. We relate those write and read operations to X and Y with the *reads-from* (rf) dashed green arrows. For instance, in the first two execution, a reads Y's initial value 0 because it is not yet overwritten. In contrast, in the right-most execution, a observes the value of Y after it is assigned 1.

Weak Behavior. Most computer architectures show additional *weak* behaviors where the order among operations within a thread is not preserved. For our Message-Passing program, we could thus observe both threads executing its operations in reverse, as the following diagram demonstrates with the solid blue arrows:



(MP-weak)

The rf arrows point from written values to subsequent reads from the same locations along the global order. We thus reach the final assignment where `a=1` and `b=0`. Crucially, that outcome is *impossible* when threads only interleave. Only when instructions execute out-of-order can we observe this outcome, making it a *weak* behavior.

The MP-weak behavior is observable on Arm (Alglave et al., 2021), but not on x86 (Owens et al., 2009). As these architectures have different weak memory models, they preserve different orders within each thread. Arm reorders the write-write pair on the first thread and the read-read pair on the second; either reordering alone could also produce the weak outcome. Those pairs *cannot* reorder on x86, making this behavior impossible on x86. If we had naively translated the program from x86 to Arm, the MP-weak behavior with outcome `a=1,b=0` could unexpectedly appear.

Robustness. Although Arm preserves the order among fewer memory operations than x86, it provides memory primitives to *explicitly* order them. For instance, we can place *fences* in our Message-Passing program as follows:

```
       Arm
    X=0, Y=0
X = 1;  ||  a = Y;
DMBFF;  ||  DMBFF;
Y = 1;  ||  b = X;
```

An Arm `DMBFF` fence, short for "Data Memory Barrier Full Fence", explicitly prevents memory operations from reordering across it. In particular, as operations no longer reorder, the MP-weak outcome is not possible anymore. For this program, we have restricted its behaviors on Arm to those on x86 – known as enforcing x86-Arm *robustness* (Chakraborty, 2021; Bouajjani et al., 2013a). When executing an x86-Arm robust program on Arm, it shows only those behaviors also observable on x86.

When migrating programs from x86 to Arm, we must enforce x86-Arm robustness to prevent new weak behaviors from appearing. A simple approach inserts

DMBFF fences between *all* adjacent memory accesses – thus enforcing sequential consistency. However, fences incur a runtime overhead (Liu et al., 2020), which varies with their location in the program and the complexity of CPU-specific bus architecture. In any case, runtime performance benefits from placing them judiciously.

PROBLEM STATEMENT

Translating programs between architectures is important in practice, as the landscape of computing hardware has evolved within the last decade, with a shift from the dominating x86 architecture (Intel Corporation, 2025) to new Instruction Set Architectures (ISAs), such as Arm (Arm Limited, 2024) and RISC-V (RISC-V International, 2024). Arm CPUs are now widely deployed, on both consumer and enterprise hardware, for instance, with Apple's M3 (Apple Inc., 2023), Google Axion (Vahdat, 2024), AWS Graviton (Stormacq, 2022), and Microsoft Cobalt (Kishan and Borkar, 2024). When naively translating legacy x86 programs to Arm, they could unexpectedly show new weak behaviors, of which MP-weak is an example. We thus ask:

Research Question

How can we correctly translate programs between weak memory model architectures while minimizing performance overhead?

We approach this problem from multiple angles, each with its own correctness and performance characteristics. The primary contexts within which we consider this problem are *binary translation with mapping schemes* and by *dynamic program analysis*.

Binary Translation with Proven Mapping Schemes. When the source code of the legacy program –or a part of it– is no longer available or when it contains architectural intrinsics, it cannot simply be recompiled. *Binary translation* addresses this problem by translating the compiled binary program (*e.g.*, from x86) to another architecture (*e.g.*, to Arm). That translation process should be fast, as it takes place shortly before executing the program and thus directly affects the time needed to execute the program to completion. Hence, a binary translator often cannot afford to thoroughly analyze the program. Instead, we aim to define *mapping schemes*, which translate programs instruction-by-instruction. These mapping schemes should be *correct in general*, for any program with any number of threads, but also minimize performance overhead. In particular, our correctness criterion concerns *robustness*, where the mapped program must not show additional weak behaviors on the new architecture – about which we thus ask:

Sub-Question 1

How can we define performant *mapping schemes* between weak memory model architectures and prove these preserve robustness?

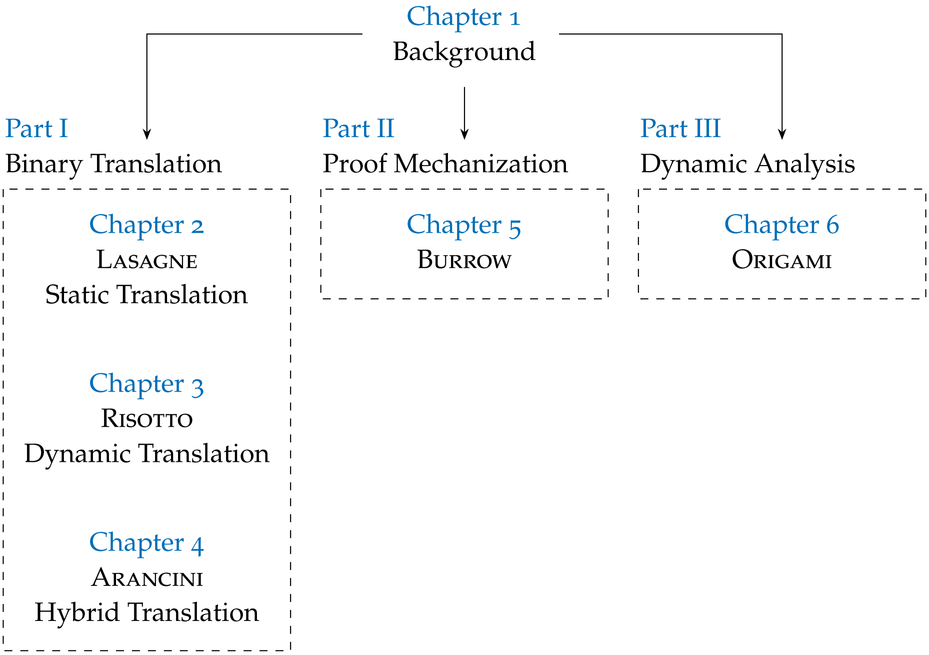
Dynamic Program Analysis. Although the mapping schemes are a *general* solution, useful when we *cannot* analyze the program, they are often not optimal for a *specific* program. However, when we know the program, we could analyze it to place fewer fences that still enforce robustness. In particular, we can fix only those violations that appear in program traces when simulating its execution – about which we ask:

Sub-Question 2

How can we detect and fix robustness violations that appear in program traces?

DISSERTATION STRUCTURE

We answer our research questions in the main chapters of this dissertation, which are based on five papers. The presentation within those papers differs from this dissertation. Although the articles are internally complete, their background and formal definitions overlap considerably, and are thus extracted into "Chapter 1 – Background". We visualize the relation between the chapters as follows:



For the papers included in Part I, I am *co-lead* author and was responsible for the formal semantic models, mappings, transformations, and mechanized proofs. Within this dissertation, contributions of my collaborators are reduced.

The remainder of this introduction summarizes the parts with their corresponding chapters. In Part I we propose novel formal weak memory models for various binary translation systems, with mapping schemes to translate from x86 to Arm and RISC-V, each accompanied by mechanized proofs in Agda (Agda Team, 2025a). Through this formalization, we identified errors in existing memory models and program translators. Notably, we propose a fix to the Arm memory model in Chapter 3. In Part II we elaborate on the proof engineering challenges we faced when mechanizing the proofs for our mapping schemes in Part I and propose a general approach to simplify writing such proofs in Agda. Finally, in Part III we introduce an alternative method to enforce x86-Arm robustness by dynamically analyzing any program to fix only those violations that it actually observes at runtime.

PART I: BINARY TRANSLATION

In part I we look at *binary translation*, where we translate compiled binary programs from one architecture to another. The contained Chapters 2 to 4 present different approaches to translate binary programs, each with its own challenges. Before going into their individual challenges, we explain their shared context and background.

With binary translation we face the robustness challenges as explained above, but *additionally* require the translation itself to be fast because it takes place shortly before executing the program. That requirement rules out existing approaches that thoroughly analyze the program to enforce robustness (Chakraborty, 2021; Bouajjani et al., 2013a; Alglave et al., 2017), which are too computationally demanding within binary translators.

Mapping Scheme. Our approach enforces x86-Arm robustness with *mapping schemes* that translate x86 memory instruction-by-instruction to Arm with appropriate fences. On page 11 we showed an example where we added *full* DMBFF fences between adjacent instructions for the Message-Passing program. However, that approach would order many Arm programs *stronger* than the original x86 program, unnecessarily harming the program's runtime performance.

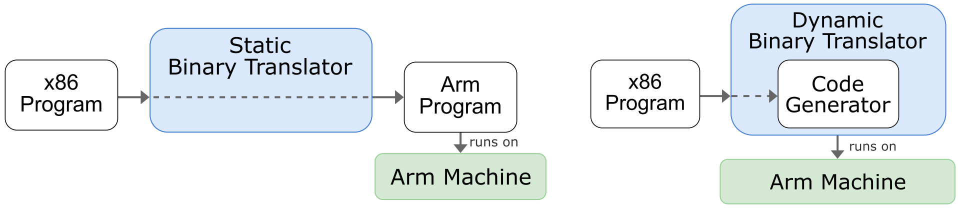
Instead, we can insert lightweight fences. While a full fence orders *any* memory accesses across it, Arm's load fence DMBLD orders only preceding loads with any succeeding accesses, while its store fence DMBST orders only stores across it. Those lightweight fences incur a lower runtime overhead than full fences (Liu et al., 2020), but can preserve x86-Arm robustness when placed correctly. We thus define a mapping scheme resembling the following, which is simplified[1] for presentation:

x86		Arm
ld	→	ld;DMBLD
st	→	DMBST;st

[1] RMW and fence instructions are omitted.

When translating from x86 to Arm, we place a load fence after every load instruction (`ld`) and a store fence before every store instruction (`st`). However, we can only know the mapping scheme is correct by ensuring any behavior of the resulting Arm program was also observable for original x86 program. Hence, we need to carefully inspect the weak memory semantics of both architectures (Owens et al., 2009; Alglave et al., 2021) to formally prove our mapping scheme correct. For the chapters in part I, we developed mapping schemes for various binary translators, with corresponding mechanized proofs in the Agda proof assistant (Agda Team, 2025a).

Static vs. Dynamic Binary Translation. In Chapters 2 to 4, we develop such mapping schemes for three different binary translators, whose system architectures differ. Binary translators are often either *static*, which translate the entire program *before* running it, or *dynamic*, which translate it *while* running. Figure 2 illustrates those architectural differences. Either approach has its own strengths and weaknesses. By translating a program before execution, a static translator can apply whole-program optimizations, benefiting performance. However, static translation is undecidable in general (Rice, 1953) –meaning a static binary translator that can translate every program *cannot exist*– because much semantic information was lost during compilation (Andriesse et al., 2016). In contrast, by translating the program while executing, a dynamic binary translator does not need to recover all information beforehand—which is impossible in general—and thus only recovers the information needed in its specific runtime context. Unfortunately, that runtime translation often incurs a greater performance overhead than static translation. Finally, a *hybrid* binary translator has characteristics of both static and dynamic, thus gaining benefits from both.



(a) Static Translator (b) Dynamic Translator

Figure 2: High-level Architectures of Binary Translators. In either case, the objective is to execute an x86 program on Arm, but the approaches differ. A static translator translates the entire program, which then executes on the Arm machine. In contrast, a dynamic translator continuously translates program *fragments* "just-in-time", as they are encountered at runtime.

We will now explain the individual contributions of the chapters in part I.

Chapter 2: Static Program Translation with Mapping Schemes

> Based on "*Lasagne: A Static Binary Translator for Weak Memory Model Architectures*" by Rodrigo Rocha*, Dennis Sprokholt*, Martin Fink, Redha Gouicem, Tom Spink, Soham Chakraborty, Pramod Bhatotia, at PLDI 2022 (* = co-lead authorship)

Chapter 2 introduces our *static* binary translator Lasagne, which builds upon the LLVM compiler framework (Lattner and Adve, 2004) and mctoll lifter (Yadavalli and Smith, 2019), which *lifts* x86 binary programs to the LLVM Intermediate Representation (IR). Lifting aims to reverse the compilation process from LLVM IR to x86 through which the original x86 binary program was produced. After lifting the programs, it naturally uses LLVM's existing compilation backend to produce the final Arm binary program. We identified robustness errors in the existing translation by LLVM and mctoll, which we fixed in Lasagne with our formally verified mapping scheme.

We first define a formal LLVM IR Concurrency Memory Model (LIMM), which is intermediate within the mapping scheme. As an example, the simplified variant of our mapping scheme becomes:

x86		LIMM		Arm
`ld`	→	$\texttt{ld};\texttt{F}_{\texttt{rm}}$	→	`ld;DMBLD`
`st`	→	$\texttt{F}_{\texttt{ww}};\texttt{st}$	→	`DMBST;st`

LIMM defines the semantics of our load fence $\texttt{F}_{\texttt{rm}}$, which orders preceding reads with any succeeding memory operation, and our store fence $\texttt{F}_{\texttt{ww}}$, which orders preceding writes with succeeding writes. Although this mapping goes through LIMM, the composed x86-to-Arm mapping is unaffected. The modular mappings through LIMM allow us to optimize the intermediate LLVM IR program before producing the final Arm binary. Existing LLVM optimizations regularly remove *false dependencies*, such as the peephole optimization '$v \times 0 \mapsto 0$'. Removing v may affect the program's reordering behavior on Arm (Pulte et al., 2017), thus invalidating such optimizations. To avoid those challenges, we *do not order* memory accesses based on dependencies in LIMM, but only explicitly with fences. We proved several optimizations on LIMM, such as reordering and elimination of various memory instructions, which are commonly used in existing LLVM optimizations.

Within our memory model and mappings we also carefully consider Read-Modify-Write (RMW) operations, which atomically read and write to shared memory. An `RMW` *fails* when another thread concurrently writes to the same memory location. The success case may enforce subtly different orders from the failure case, depending on the ISA. Additionally, while x86 has single-instruction `RMW` instructions, on Arm these are often implemented with two separate –but connected– *load-linked store-conditional* (ll/sc) instructions, which also order weaker (Pulte et al., 2017; Alglave et al., 2014). Our mapping scheme correctly translates x86's `RMW` to Arm's corresponding instructions, for both the success and failure cases, by placing appropriate fences.

Chapter 3: Dynamic Program Translation with Mapping Schemes

> Based on "*Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures*" by Redha Gouicem*, Dennis Sprokholt*, Jasper Ruehl, Rodrigo Rocha, Tom Spink, Soham Chakraborty, and Pramod Bhatotia, at ASPLOS 2023 (* = co-lead authorship)
>
> Recipient of an ASPLOS Distinguished Artifact Award

Chapter 3 introduces our *dynamic* binary translator RISOTTO, which builds upon the existing QEMU dynamic binary translator (Bellard, 2005; QEMU Team, 2003). QEMU internally uses its Tiny Code Generator (TCG), with TCG IR language. QEMU does not officially support strong-on-weak ISA execution, such as running x86 programs on Arm.

We observed QEMU's existing translation attempts to enforce a *stronger* ordering than x86 when executing on Arm, unnecessarily hurting performance (Liu et al., 2020). Despite those attempts, it still fails to translate programs correctly, as we discovered several translation errors. We eliminate those errors by defining the first formal weak memory model of TCG IR, called TIMM, and proving the mappings correct from x86 to TCG IR to Arm. We replaced the existing erroneous mappings in QEMU with our verified mappings. Additionally, we proved several optimizations correct on TCG IR, such as memory access eliminations, which commonly appear in TCG's constant propagation and folding passes, as well as various reordering optimizations.

We also proved our translations correct with respect to a newer Arm model (Alglave et al., 2021) than we did for LASAGNE (Chapter 2). This new model specifically includes formal semantics for a new Arm "Compare and Swap Acquire Release" $\mathtt{CAS}_{\mathtt{AL}}$ instruction, which is a *single-instruction* RMW operation, unlike Arm's prior ll/sc pair. That new $\mathtt{CAS}_{\mathtt{AL}}$ aims to "act as a full barrier" (Alglave et al., 2021, p.18), like x86's RMW. However, through our formal proofs we discovered that it orders weaker than a full barrier! To correctly translate x86 programs, we thus must either *(i)* place strong fences around the $\mathtt{CAS}_{\mathtt{AL}}$, like we did with ll/sc pairs in Chapter 2, thus harming performance; or *(ii)* strengthen the Arm model. As the former option would invalidate the advantage of $\mathtt{CAS}_{\mathtt{AL}}$ over ll/sc, we prefer the latter, for which we propose a fix to the Arm model. After reporting the error to the authors, they included an equivalent strengthening (Alglave, 2022).

Chapter 4: Hybrid Program Translation with Mapping Schemes

> Based on "*Arancini: A Hybrid Binary Translator for Weak Memory Model Architectures*" by Sebastian Reimers*, Dennis Sprokholt*, Martin Fink, Theofilos Augoustis, Simon Kammermeier, Rodrigo Rocha, Tom Spink, Redha Gouicem, Soham Chakraborty, and Pramod Bhatotia (* = co-lead authorship)

Chapter 4 introduces our *hybrid* binary translator Arancini. Static translators cannot translate all programs (Andriesse et al., 2016) because semantic information was lost when compiling the original program, which cannot be recovered in general (Rice, 1953). For instance, lifting compiled C++ programs is challenging as they commonly use dynamic dispatch, where jump targets are known only at runtime, which we often cannot recover statically. In contrast, dynamic translation can translate any program but incurs a performance overhead at runtime because the translation happens during program execution. Through hybrid translation, Arancini gains the benefits of both. Arancini translates much of the program statically, thus avoiding the runtime performance overhead, while resorting to dynamic translation whenever static translation is impossible.

Again, we define and prove mapping schemes that extend upon Lasagne (Chapter 2) and Risotto (Chapter 3) in *two* ways: *(i)* by translating *mixed-size* accesses (Flur et al., 2017; Alglave et al., 2021) from x86 to Arm, which access memory in units of multiple size (*e.g.*, 1/2/4/8-byte units); and *(ii)* by translating x86 to RISC-V (RISC-V International, 2024). For Arancini we define ArancinIR, a custom IR with corresponding weak memory model AIMM, resembling Lasagne's LIMM and Risotto's TIMM models but additionally includes semantics for mixed-size accesses.

During our proof efforts, we discovered *splitting transformations* are incorrect. For instance, it is incorrect to split a 16-bit store operation into two separate 8-bit stores, as the example demonstrates below. The 16-bit variable `X` –initially `0`– consists of two adjacent 8-bit variables $\texttt{X}_H$ and $\texttt{X}_L$, denoting `X`'s most and least significant byte, respectively.

$$\begin{array}{l||l} \texttt{X} \overset{16}{=} \texttt{0x1234;} & a \overset{8}{=} \texttt{X}_L; \\ & \texttt{F}_{\texttt{rm}}; \\ & \texttt{X}_H \overset{8}{=} \texttt{0xFF;} \end{array} \quad \not\rightarrow \quad \begin{array}{l||l} \texttt{X}_L \overset{8}{=} \texttt{0x34;} & a \overset{8}{=} \texttt{X}_L; \\ \texttt{X}_H \overset{8}{=} \texttt{0x12;} & \texttt{F}_{\texttt{rm}}; \\ & \texttt{X}_H \overset{8}{=} \texttt{0xFF;} \end{array}$$

While the original program *cannot* show a terminal state with `X`=`0x1234` and a=`0x34`, the split erroneously introduces that unforeseen behavior; when the second thread executes entirely in-between the instructions on the first thread. As the error already appears when only interleaving threads, it also appears in any weaker memory model.

PART II: PROOF MECHANIZATION

Part I has introduced the formal memory models for the languages inside our binary translators, with mapping proofs between architectures and optimization proofs. As the formal axiomatic weak memory models we used are often large and complex, writing such proofs requires inspecting many cases with subtle complexities that are often error-prone (Batty et al., 2016; Manerkar et al., 2016; Sarkar et al., 2012, 2011). To avoid missing proof cases, we *mechanized* their proofs in the Agda proof assistant (Agda Team, 2025a), significantly increasing their reliability.

The proofs for Lasagne, Risotto, and Arancini independently consist of *thousands* of lines of Agda, making them large and especially *rigid* code bases. As with any large computer program's source code, there are many ways to structure the code, some more maintainable than others. Throughout our proof engineering efforts for mapping schemes, we tried numerous proof structures to reduce their complexity and increase maintainability. Within part II, we present the results of those efforts.

Chapter 5: Weak Memory Mapping Proofs in Agda

> Based on "*Mechanizing Weak Memory Proofs in Agda*"
> by Dennis Sprokholt and Soham Chakraborty

Chapter 5 introduces our Agda proof framework Burrow to *mechanize* axiomatic weak memory semantics and prove corresponding transformations correct. Although existing approaches bring significant theoretical contributions – for instance, by finding errors in existing models (Chakraborty and Vafeiadis, 2016, 2017), repairing those models (Lahav et al., 2017), or introducing new models (Batty et al., 2011; Kang et al., 2017) – writing the corresponding mechanized proofs remains time-consuming for proof engineers. Burrow reduces those proof mechanization challenges with custom primitives and abstractions that are specific to the domain of weak memory transformation proofs.

Burrow leverages Agda's specific strengths, which differ from those of other proofs assistants. Whereas idiomatic Rocq (Bertot and Castran, 2010) favors extensive use of *tactics* to automatically solve goals, Agda's explicit term manipulation with dependent pattern matching (Coquand, 1992; Cockx and Abel, 2018) and easy syntax extension –but lack of tactics– emphasizes using the right formalization abstractions. In particular, we define domain-specific proof primitives, with which Burrow proves large fragments of weak memory transformation proofs *generally*.

We first define the high-level theorem, which all our proofs follow. For the Message-Passing example on page 10, we saw that a program may show one of multiple executions, depending on thread interleaving and out-of-order execution. However, Arm can show the MP-weak behavior, which x86 does not. If we had naively translated the program from x86 to Arm, that additional behavior would suddenly appear, which is incorrect. Instead, we must ensure that any behavior observed when executing the target program $\mathbb{P}^{\mathrm{Arm}}$, was also a behavior the source program $\mathbb{P}^{\mathrm{x86}}$, which we can visualize below. $\mathsf{G}^{\mathrm{x86}}$ denotes the set of behaviors of the source program $\mathbb{P}^{\mathrm{x86}}$, while $\mathsf{G}^{\mathrm{Arm}}$ denotes the set of behaviors of the target program $\mathbb{P}^{\mathrm{Arm}}$.

$$\begin{tikzcd}
\mathbb{P}^{\mathrm{x86}} \arrow[r, "\text{map}"'] \arrow[d, "\text{execute}"'] & \mathbb{P}^{\mathrm{Arm}} \arrow[d, "\text{execute}"] \\
\mathsf{G}^{\mathrm{x86}} \arrow[r, phantom, "\supseteq"] & \mathsf{G}^{\mathrm{Arm}}
\end{tikzcd}$$

After defining this proof structure in Agda, we define its components, which ensure executions are well-formed and consistent with their architecture's weak memory semantics. This same structure applies when proving optimizations correct.

Next, we identify and clarify ambiguities in general weak memory model definitions. Although they are formal models, much of their definitions are given only *implicitly*. While pen-and-paper proofs may permit unspecific or ambiguous definitions, mechanized proofs demand meticulous specification of all details. The mechanized Agda definitions thus precisely capture the weak memory semantics. Using those definitions, Burrow provides several mapping primitives and abstractions with which programmers only need to *define* the mapping between architectures and prove that they preserve several simple properties. Burrow then produces large fragments of the weak memory proofs generally, allowing proof engineers to focus only on the interesting parts.

We demonstrate Burrow's effectiveness by proving the conjectured mapping from x86 to Arm by Alglave et al. (2021, §2.5), which is only true after fixing the Arm model (Alglave, 2022), as discussed in Chapter 3. Their mapping differs from ours (Chapters 2 to 4), as ours insert fences around memory accesses, while theirs annotate accesses with memory orders – which is an alternative method to preserve memory orders on Arm. We prove their alternative mapping as a case study to demonstrate Burrow's versatility.

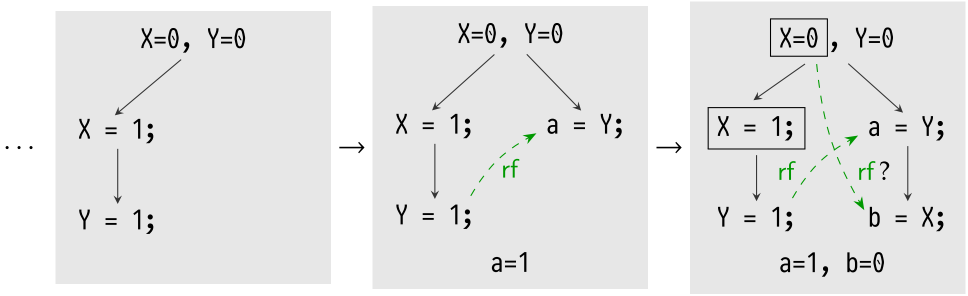
Part III: Dynamic Analysis

In part I and II we looked at mapping schemes to enforce x86-Arm robustness. The advantage of mapping schemes is that they are correct *for any program* and require no expensive program analysis to determine fence placement, which was crucial for the binary translators in part I. However, their primary disadvantage is that they may insert excessively many fences, which we could avoid when we know the specific program. In part III we aim to analyze any program to enforce memory orders only when needed.

Chapter 6: Porting Programs with Dynamic Analysis

> Based on "*Porting Concurrent Programs between Weak Memory Architectures*" by Dennis Sprokholt, Anish Yogesh Kulkarni, Yifan Song, S. Krishna, and Soham Chakraborty, under submission

Chapter 6 introduces our dynamic program analyzer Origami, which identifies x86-Arm robustness violations by simulating programs step-by-step under both the x86 and Arm memory model, reporting mismatching steps. Consider again the Message-Passing program, where the order of instructions is preserved on both threads in x86, but on Arm they can reorder. We construct one execution step-by-step, where at every step we select the next instruction to execute at random, shown after the first *two* steps:



Unlike with the previous execution examples (on page 10), where we showed a *global* blue order among events, we now only show the black *program order* (po) arrows among operations *within each thread*. Upon appending the final read operation from X, we must select a write operation that it reads-from (rf). As the preceding operation on the second thread already observed the value 1 being written to Y, it established a causal order between them on x86. On Arm, the operations on both threads can freely reorder, making it possible to observe X=0 in the final read operation. In contrast, x86 *cannot* observe that value, as it is "hidden behind" the assignment X=1. Hence, we have discovered a robustness violation, where Arm shows the additional behavior where a=1,b=0, which is impossible on x86. That violation is the same as the MP-weak execution seen before.

To analyze programs step-by-step, we first define an operational-axiomatic weak memory semantics for x86 and Arm, based on existing axiomatic semantics (Owens et al., 2009; Alglave et al., 2014, 2017). These semantics formally define the valid program steps in both architectures. Using those semantics we define an x86-Arm robustness analysis algorithm, which repeatedly takes a step in both models at random. We implement the analysis algorithm in Origami, which builds upon C11Tester (Luo and Demsky, 2021) to randomly explore program executions. Upon encountering a robustness violation, C11Tester reports the program trace that produced it.

After discovering a robustness violation, we need to fix it. As part of Origami, we define another robustness *enforcement* algorithm, which identifies all weak paths leading to the violation. For instance, in the above example, it identifies the black po edges between the operations in both threads. As those operations are ordered in x86 but not in Arm, the orders on both threads must be strengthened on Arm, for instance, by inserting fences. To do that, our enforcement algorithm proposes minimal strengthenings to prevent a given robustness violation from appearing again.

After several iterations of analysis and enforcement, Origami finds no further robustness violations. In contrast to existing model checkers (Abdulla et al., 2015a,b; Bouajjani et al., 2013a; Oberhauser et al., 2021), whose *exhaustive* exploration of the program's state space often does not scale, Origami's random exploration makes it very suitable for large programs. Unfortunately, either approach may miss violations: Model checkers miss violations beyond their reachable scale, while Origami misses those unexplored by its random process. However, we experimentally demonstrate that Origami identifies many x86-Arm robustness violations in existing benchmark programs (Luo and Demsky, 2021; Chakraborty, 2021) while inserting fewer fences than our mapping schemes (Chapters 2 to 4), benefiting the program's runtime performance.

PAPERS

The work presented in this dissertation is also included in the following papers (not all of which are published), listed in reverse chronological order:

Mechanizing Weak Memory Proofs in Agda
Dennis Sprokholt, Soham Chakraborty

Porting Concurrent Programs between Weak Memory Architectures
Dennis Sprokholt, Anish Kulkarni, Yifan Song, S. Krishna, Soham Chakraborty

Arancini: A Hybrid Binary Translator for Weak Memory Model Architectures
Sebastian Reimers, Dennis Sprokholt*, Martin Fink, Theofilos Augoustis, Simon Kammermeier, Rodrigo Rocha, Tom Spink, Redha Gouicem, Soham Chakraborty, Pramod Bhatotia*

Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures
Redha Gouicem, Dennis Sprokholt*, Jasper Ruehl, Rodrigo Rocha, Tom Spink,*
Soham Chakraborty, Pramod Bhatotia (ASPLOS 2023)
🏆 **Recipient of an ASPLOS Distinguished Artifact Award**

Lasagne: A Static Binary Translator for Weak Memory Model Architectures
Rodrigo Rocha, Dennis Sprokholt*, Martin Fink, Redha Gouicem, Tom Spink,*
Soham Chakraborty, Pramod Bhatotia (PLDI 2022)

* = co-lead authorship

During my PhD research I have also collaborated on the following papers, which are *not* included in this dissertation:

Distributing Time-Affected Compute-Intensive Programs
Dennis Sprokholt, Soham Chakraborty

Cage: Hardware-Accelerated Safe WebAssembly Programs
Martin Fink, Dimitrios Stavrakakis, Dennis Sprokholt, Soham Chakraborty, Jan-Erik Ekberg, Pramod Bhatotia (CGO 2025)

TOOLS AND PROOFS

All our tools and proofs are available as open-source software.

- Lasagne (Chapter 2)
  - Proofs – github.com/binary-translation/lasagne-proofs
- Risotto (Chapter 3)
  - Proofs – github.com/binary-translation/risotto-proofs
- Arancini (Chapter 4)
  - Proofs – github.com/sourcedennis/arancini-proofs
- Burrow (Chapter 5)
  - Burrow – github.com/sourcedennis/agda-burrow
  - Dodo – github.com/sourcedennis/agda-dodo
  - Case Study – github.com/sourcedennis/armed-proofs
- Origami (Chapter 6)
  - Robustness Analysis – github.com/sourcedennis/c11tester-x86-arm
  - Robustness Enforcement – github.com/sourcedennis/enforce-robustness
  - Custom Fency Programs – github.com/sourcedennis/fency-programs

Chapter 1

Background

In this chapter, we introduce weak memory consistency semantics, which we use in the remaining chapters. First, in Section 1.1 we explain *by example* the subtle behaviors that appear on weak memory architectures. Second, in Section 1.2 we introduce the formal notations for axiomatic weak memory models, which we later use to formalize weak memory behaviors. Third, in Section 1.3 we cover established weak memory models of existing ISA architectures, such as x86, Arm, and RISC-V. Finally, in Section 1.4 we give the high-level structure of the theorems we consider in the remaining chapters, which captures the correctness criterion of transformations involving concurrency primitives.

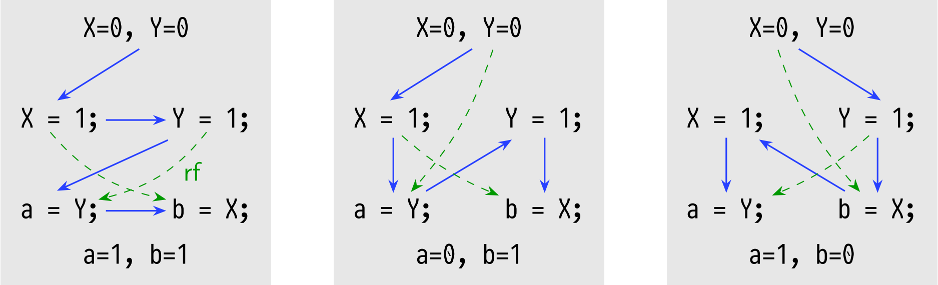
1.1 AXIOMATIC WEAK MEMORY BY EXAMPLE

We first introduce axiomatic weak memory semantics by example. On page 10 we already showed the Message-Passing program, whose behavior differs between x86 and Arm. Here, we give another program, which shows *the same* weak behavior on x86 and Arm. The program again has two threads, where X and Y are shared between them and are initialized to 0, while a and b are thread-local to their respective threads:

```
     X=0, Y=0
X = 1;  ||  Y = 1;
a = Y;  ||  b = X;
```

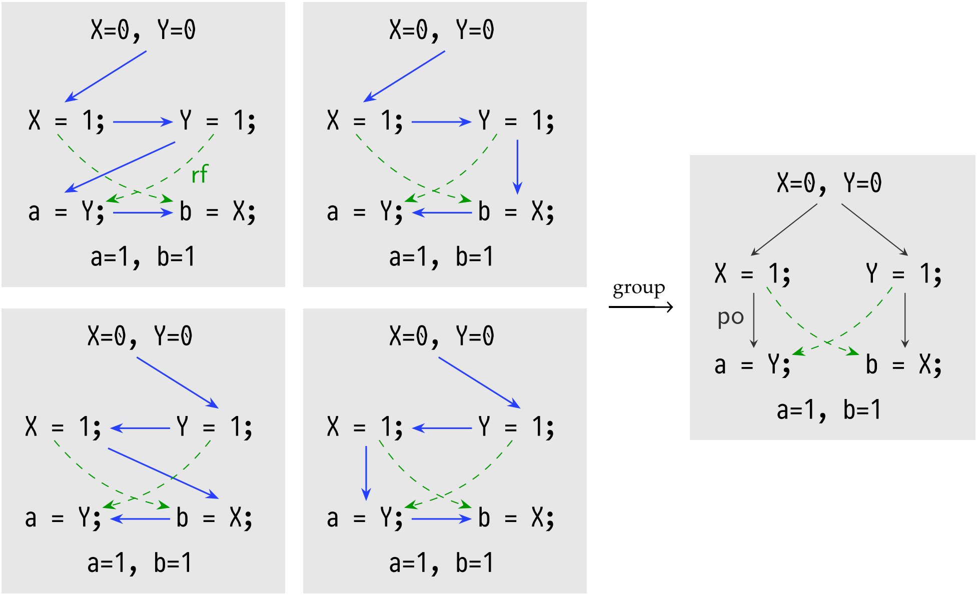
(Store-Buffering)

This program is commonly known as the Store-Buffering program because some architectures *buffer the stores* (Sewell et al., 2010), thus not immediately communicating written values to other CPU cores, resulting in weak behaviors. Before going into those weak behaviors, we look at the behaviors that emerge *only* from interleaving, which thus respect the instruction ordering within each thread. For instance, we may non-deterministically observe one of the following interleaving orders, marked with blue *solid* arrows:



Although the order among instructions within each thread is preserved in the above executions, the threads interleave arbitrarily. The green dashed *reads from* (rf) arrows relate write operations to the subsequent same-location read operation in the interleaving order. Depending on the specific interleaving, those read operations observe a value written by a different write operation, resulting in distinct assignments to a and b upon termination. Note that the outcome a=0,b=0 is *impossible* under sequential consistency.

Grouping Behaviors. The previous examples explicitly showed a global runtime order among instructions with the blue arrows. Such global orders are easily defined when considering only interleaving. However, for weak behaviors on CPUs, defining a total order among events becomes challenging – for instance, when an instruction's effects are not immediately observable to other threads. To avoid that challenge altogether, we do *not* explicitly represent a global order among events, but group executions based on the runtime interactions between instructions. To illustrate that grouping, consider the following interleaving executions for Store-Buffering:



We see four distinct interleavings, where reading from X and Y observes the value written by the same respective write operation. Consequently, each results in the same final state where a=1 and b=1. For our purposes of modeling weak memory semantics, we need not distinguish between the specific interleavings, but only consider the rf relations between the instructions, which the rightmost grouped execution shows. In that execution, we do *not* model the blue global order, but only the thread-local *program order* (po) between the instructions and the reads-from (rf) edges between them. The program order captures the runtime order among instructions as they appeared in the program's syntax.

Architectural Intuition. For the Store-Buffering program we could observe weak behaviors when a microarchitecture has *store buffers* (Sewell et al., 2010). Conceptually, a store buffers is a FIFO queue residing at each CPU core containing the pending writes to main memory, as depicted in Figure 3. As writing to memory is time consuming, these buffers allow the CPU core to continue executing other instructions while delaying the writes to main memory until later.

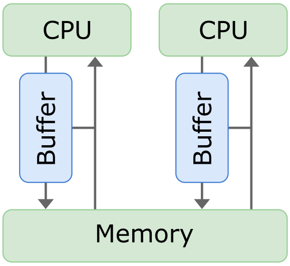
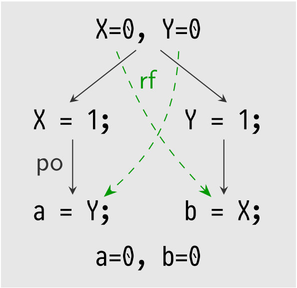


Figure 3: A depiction of *store buffers*. When a CPU core writes a value, it initially remains in a local store buffer. Those buffered values only later flush to main memory, meaning other cores cannot immediately observe them. When reading a value, a CPU core only looks in main memory when the value is absent in its local store buffer.

Store buffers illustrate why it is challenging to determine an order between instructions at runtime. Executing a write instruction is not immediately visible to other threads, as a written value remains in a local buffer. Only at a later point does it propagate from that buffer to main memory. Micro-architecturally, executing a write instruction and then propagating that value to memory are independent actions, while they are a single instruction in the program syntax. If we had explicitly modeled those actions, the global order would have gotten excessively complex. Instead, we *don't* model a global order, but only consider whether an instruction's effects is observable to another – regardless of how that took place inside the CPU.

Weak Behavior. As a result of store buffers, we can observe a weak behavior for the Store-Buffering program. Consider the two CPU cores writing the value 1 to variables X and Y, but each remains locally in a store buffer. When the threads respectively read the values from Y and X –which is only kept in the opposing thread's store buffer– both retrieve the value 0 from main memory. The corresponding execution is:



(SB-weak)

Note again that we do not model a global order among instructions, but only relate the write instructions to read instructions that observe their values, regardless of how that happened within the microarchitecture. Crucially, observe that this behavior where a and b read 0 from Y and X, respectively, is *impossible* when only interleaving threads. That is, there *exists no* thread interleaving producing this behavior, meaning it is a *weak* behavior.

Architecture Behaviors. The SB-weak behavior of the Store-Buffering program is observable on both x86 (Owens et al., 2009) and Arm (Alglave et al., 2021). In contrast, the MP-weak behavior (on page 11) of Message-Passing is observable on Arm but *not* on x86. In general, Arm preserves fewer orders among regular memory accesses than x86 within each thread, making Arm a strictly *weaker* architecture. Although x86's weak behaviors can fully be described as a consequence of store buffers (Owens et al., 2009), Arm shows additional weak behaviors that cannot be explained only by store buffers (*e.g.*, the MP-weak). Those weak behaviors appear because Arm can execute many instructions out-of-order (Alglave et al., 2014).

The memory consistency semantics of a particular architecture or language are *defined by* the behaviors observable for any program. Although we gave several example behaviors for two programs, which vary between architectures, we require a more general mechanism to define behaviors for any program. We can only do that with a mathematically rigorous definition of these semantics, which we explain next.

1.2 AXIOMATIC WEAK MEMORY, FORMALLY

The previous subsection gave intuitions and examples to illustrate the intricacies of weak memory concurrency. In this subsection we expand on those idea with the standard formal axiomatic weak memory semantics (Alglave et al., 2014; Batty et al., 2011), which restricts program behavior with axioms.

Purpose of Semantics. The purpose of formal semantics is to *mathematically* –without ambiguity– assign meaning to programs (Floyd, 1993). In particular, we consider *axiomatic semantics*, as originally coined by Hoare (1969), where we describe a program's behaviors with logical predicates. Crucially, semantics *only describe* program behaviors, which is *orthogonal* to their computational characteristics. Rice (1953) famously showed non-trivial semantic properties[2] are undecidable, meaning we cannot determine whether a given property holds for any program. In this section, we intend only to precisely describe the weak memory behaviors of programs.

1.2.1 *Executions*

Axiomatic weak memory semantics (Alglave et al., 2014; Batty et al., 2011) define the behavior of any program with *execution graphs*, each consisting of *events* and various *relations*. Each event is either a read (R), write (W), or fence (F) event. The R and W events have a location and value, for instance, R(X, 2) reads the value 2 from location X.

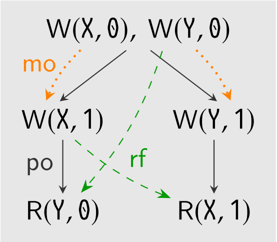
[2] An example of a semantic property is whether a given program ever reads the value 1 from some location X. It is impossible to *decide* that for *all* programs.

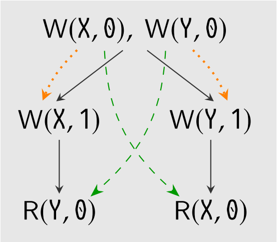
Those memory locations are concrete values (*e.g.,* 0x12345678), but we denote them *symbolically* (*e.g.,* X and Y) in the examples, for clarity of presentation[3]. Those symbolic locations strictly represent *distinct* locations in memory (*i.e.,* X and Y are strictly different). Each event also has an identifier that is unique within the execution.

We capture various relations between events, of which we already gave po and rf by example in Section 1.1. We distinguish between *primitive* and *derived* relations. The latter follow entirely from the former, but serve as notational conveniences. The typical *primitive* relations are:

- Program order (po), relating events by their syntactic order: the order among instructions in the program text carries over to execution graphs with po edges. It is a total order per thread.
- Reads-from (rf), which relates a W event to the R event reading from it. Every R event reads-from exactly one W event. For instance, when event e_2 : R(X, 1) reads 1 from location X, where that value was earlier written by e_1 : W(X, 1), then we have an rf edge from e_1 to e_2, alternatively denoted as rf(e_1, e_2).
- Modification order (mo), which relates a W event with a W event overwriting it. For instance, when event e_3 : W(X, 1) first writes 1 to location X, which event e_4 : W(X, 2) then *overwrites* with 2, then we have an edge mo(e_3, e_4). Both mo-related write events access the same location. Additionally, mo is a total order per location.
- Read-modify-write (rmw), relating the R and W of a successful atomic read-modify-write instruction (RMW). The RMW instructions are special instructions within architectures, where architectures can detect whether another thread modifies the written value in-between the atomic operation. The generated R and W events access the same location and are po-adjacent (*i.e.,* there are no po-intermediate events). We elaborate on these for various architectures in Section 1.3.

To illustrate those events and relations, we show two such execution graphs for the Store-Buffering program, consisting only of primitive relations.





Although the program syntax also contained thread-local variables a and b, those are omitted[4] in our semantics. After all, these executions aim to capture only the order

[3] In contrast, in the mechanized proofs of Chapters 2 to 5 and analysis of Chapter 6, locations are concrete.

[4] If we had wanted to capture the thread-local variables in the graphs, we could introduce additional shared variables A and B to which we write the values of the thread-local variables a and b.

among operations communicating through shared memory, which are X and Y for this program. The left execution captures a sequentially consistent behavior, resulting in a=0,b=1. The right execution captures a weak behavior, resulting in a=0,b=0.

The initial W events, which precede the start of all threads, capture the initial state of memory. In reality, that initial state was established before the program started –*e.g.*, by preceding programs– and is beyond the program's semantics. We always initialize those values to 0 in visual execution graphs. Finally, note that we omit transitive edges (*e.g.*, for po and mo) in the graphs for presentational clarity.

With the events and primitive relations, we define an execution:

▸ **Definition 1** (Execution). We denote an execution as $\mathsf{X} \triangleq \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{rmw} \rangle$, where X.E is the set of events, X.po, X.rf, X.mo, X.rmw are the set of po, rf, mo, and rmw relations, respectively, between events in X.E.

Derived Relations. We often use *derived* relations, which are notationally convenient but follow entirely from the primitive relations. Common derived relations are:

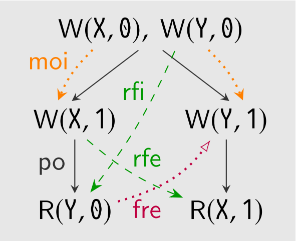
- From-read (fr), relating a R with an overwriting W event. For instance, when first reading 1 from X with e_5 : R(X, 1), which is overwritten with 2 by e_6 : W(X, 2), we have an edge fr(e_5, e_6). This relation is derived as $\mathsf{fr} \triangleq \mathsf{rf}^{-1};\mathsf{mo}$

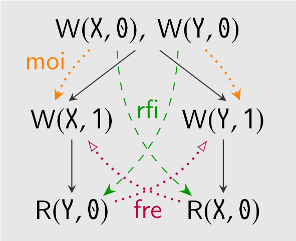
  > *Notation*[5]. rf^{-1} is the inverse of rf, *i.e.*, $\mathsf{rf}^{-1}(x, y) \triangleq \mathsf{rf}(y, x)$; while ';' represents composition of binary relations, *i.e.*, $(S_1 \,;\, S_2)(x, y) \triangleq \exists z. S_1(x, z) \wedge S_2(z, y)$.

- *External* relations are those between different threads. For instance, rfe consists of all rf edges between threads; moe is external-mo and fre is external fr. Conversely, *internal* relations are those within the same thread, such as rfi, moi, and fri. We consider the initialization W events to be internal with events on any thread.

- Same-location relations, such as $\mathsf{po}_{\mathsf{loc}}$, which relates only po-related (R and W) events that access the same memory location.

Examples. We sometimes draw derived relations. As an example, we redraw the above execution graphs with several such derived relations:





[5] We give notations inline but also give an overview in Figure 7 at the end of this chapter.

Observe that the fre edges always follow backwards up an rf edge and then forwards along an mo edge (*i.e.,* $\mathsf{fr} \triangleq \mathsf{rf}^{-1};\mathsf{mo}$), thus relating a R to a W overwriting it. For instance, in the left execution we can only read 0 from Y *before* overwriting Y with 1. All fr relations in both executions are external, producing fre arrows, while the mo relations are internal, producing moi arrows.

Behavior. When executing a program, only the values in memory are externally observable. In contrast, store buffers and mechanisms to execute instructions out-of-order are internal to a microarchitecture. We thus define the *behavior* of an execution as the final values in all memory locations.

As mo relates all write events to any particular location, per location, the mo-final event captures its final value in memory. An mo-final event is an event which has no mo-successors. Formally, we thus define an execution's behavior as:

▶ **Definition 2** (Behavior)**.** The behavior of an execution X is:

$$\mathsf{Behavior}(\mathsf{X}) \triangleq \{\langle e.\mathsf{loc}, e.\mathsf{val}\rangle \mid e \in \mathsf{X.W} \wedge [\{e\}];\mathsf{X.mo} = \emptyset\}$$

Note that $\mathsf{X.W} \subseteq \mathsf{X.E}$ is the set of *write* events in X. The notation $[P]$ represents the *identity relation* of set P, for any set P; meaning $[P](x, y) \triangleq P(x) \wedge x{\equiv}y$ [6].

1.2.2 *Program to Execution*

An execution (Definition 1) states a particular runtime behavior of a program. When running a program, it often non-deterministically demonstrates one of multiple executions, which differs upon executing it again. The *semantics* of a program is the *set* of all executions the program can show when running on an architecture.

An execution graph does not give the semantics itself, as it does not state whether or not it is observable on an architecture. However, it provides a *language* (Alglave et al., 2014) on which we *can define* semantics. For instance, although we can define an execution for SB-weak, only a semantics–external to that graph–can tell us whether the behavior is possible. In this case, sequential consistency semantics asserts SB-weak is *impossible*, whereas x86 and Arm semantics assert it *is possible*.

While we define semantics over the execution graphs, these graphs connect to the program's syntax. The program generates the execution graph upon execution, while its concurrency primitives (*i.e.,* read, write, and fence instructions) generate the events and relations between them. Within weak memory semantics, we capture *only* the memory events and relations within a program; Other parts of the program's execution, such as arithmetic or thread-local computations, are omitted. The concurrency primitives vary subtly between architectures. However, any primitive can be categorized as a read, write, or fence – or some combination thereof (*e.g.,* RMWs read *and* write). We elaborate on the specific concurrency primitives for various architectures in Section 1.3.

[6] We notationally distinguish between propositional equality '≡' and definitional equality '=', which is needed in proof assistants (Chapter 5); we must explicitly prove the former, while the latter holds trivially.

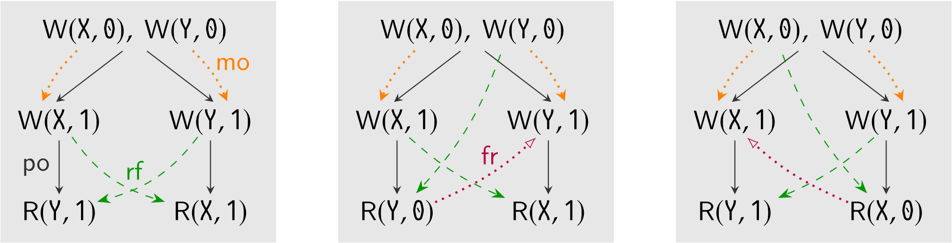
1.2.3 *Axiomatic Consistency Semantics*

With semantics we prescribe which behaviors are observable when executing a program. In particular, we precisely assert which executions are observable on a particular architecture and which are not. We do that with consistency axioms for any particular model. Executions satisfying the axioms are observable, while executions violating it are not. We alternatively call an execution that satisfies all axioms of a model *consistent* with that model. Formally, we denote the *set* of executions of a program $\mathbb{P}$ consistent with weak memory model M as $[\![\mathbb{P}]\!]_M$. Then the *behavior* of $\mathbb{P}$ under M is the set of behaviors (Definition 2) exhibited by *all* executions in $[\![\mathbb{P}]\!]_M$.

Example: Sequential Consistency. We illustrate definitions above with the semantics of sequential consistency (Lamport, 1979), where programs must satisfy[7]:

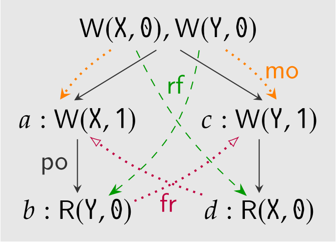
$$(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{fr}) \text{ is acylic} \qquad \text{(SC)}$$

This SC axiom states that any execution consistent with sequential consistency *must not* contain cycles consisting of po, rf, mo, and fr edges. As an example, we look at the consistent executions for the Store-Buffering program, which are $[\![\text{Store-Buffering}]\!]_{\text{SC}}$:



Those are *all* executions of Store-Buffering that are consistent with the SC axiom and thus the sequential consistency model. Observe that we presented all po, rf, mo, and fr edges; and that none of those edges together form a cycle, and therefore satisfy SC. The *behavior* of all these executions is $\{\langle \mathtt{X}, 1\rangle, \langle \mathtt{Y}, 1\rangle\}$, as 1 is the mo-final value for both X and Y in all executions. Unlike our prior presentations in Section 1.1, the formal behavior does not include the final state of thread-local variable (*i.e.*, a and b).

Of course, weak behaviors are *not* allowed by sequential consistency, of which SB-weak already presented an (informal) example. We now inspect the formal variant of that execution to see whether it actually violates the SC axiom.



(SB-weak-formal)

[7] For presentational simplicity, we ignore RMW instructions here.

Indeed, this execution contains a 'po;fr;po;fr' cycle, which traverses $a\xrightarrow{\mathsf{po}}b\xrightarrow{\mathsf{fr}}c\xrightarrow{\mathsf{po}}d\xrightarrow{\mathsf{fr}}a$. As it *violates* the SC axiom it is *inconsistent* with the sequential consistency model.

1.3 AXIOMATIC WEAK MEMORY ARCHITECTURE SEMANTICS

Although sequential consistency is a simple model, it is *too strong* to describe the behavior of many architectures and languages. As the MP-weak behavior is observable on Arm and the SB-weak behavior is observable on both x86 and Arm, their models need different weak axioms to capture those behaviors. Before going into the specific details of the architecture models, we look at their common axioms.

1.3.1 *Common Axioms*

Two primary axioms are shared between all models we consider, which are Coherence[8] and Atomicity. We present those common axioms here.

Coherence. Coherence states that accesses to the same memory location are sequentially consistent. That is a much weaker requirement than SC from before, as accesses to different locations are *not* sequentially consistent. Formally, we state this axiom as:

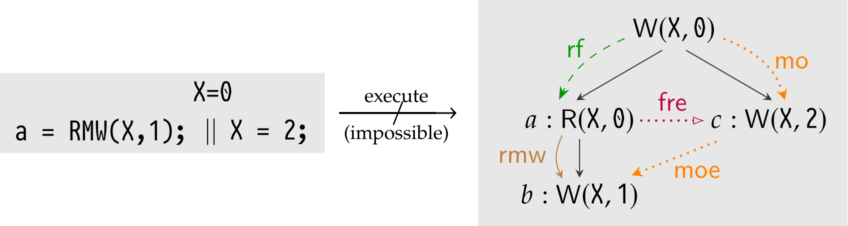
$$(\mathsf{po}_{\mathsf{loc}} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{fr}) \text{ is acylic} \qquad \text{(Coherence)}$$

Observe that the SB-weak-formal execution contains no violating cycles, as all po-edges are between events on different locations. That means the execution is *allowed* by this axiom, which contrasts the stronger SC axiom that had *disallowed* it.

Atomicity. Atomicity captures the ordering behavior of RMW instructions that exist on x86, Arm, and RISC-V. An RMW instruction atomically reads and writes from a memory location. That instruction can *fail* when another thread simultaneously writes to the same location, in which case the RMW only generates a R event. However, when it succeeds, it generates both a R and W event, related by an rmw edge. In that case, we know for certain that no other thread had simultaneously written to the same location in memory. The atomicity axiom precisely captures that property:

$$\mathsf{rmw} \cap (\mathsf{fre}\,;\mathsf{moe}) = \emptyset \qquad \text{(Atomicity)}$$

The axiom requires the intersection between rmw and 'fre; moe' to be empty. To illustrate what it disallows, consider a simple program with two threads, where a RMW instruction on the first thread writes 1 while the second threads writes 2:

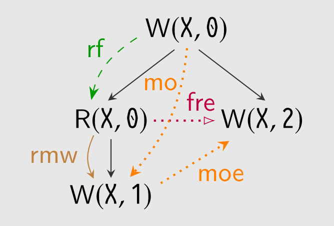


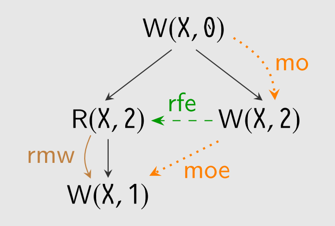
```
        X=0
a = RMW(X,1); || X = 2;
```

[8] Coherence is sometimes presented differently (*e.g.*, in the Arm model we use in Chapters 4 and 5 (Alglave and Maranget, 2025)), where it is refactored among other axioms. Coherence then still holds, but *implicitly*.

This execution is *disallowed* because it violates the Atomicity axiom. We have edges fre(a, c) and moe(c, b), which means (fre ; moe)(a, b). As we also have the edge rmw(a, b), the intersection between those is *not* empty, violating the axiom.

We also show the only two executions for the above program that satisfy Atomicity. In those executions we see no 'fre ; moe' paths between the rmw-related events. Intuitively too, the R and W events generated by the RMW instruction are treated *atomically*; no other event observes or interferes with them partially.





1.3.2 *x86 Semantics*

We now give the weak memory models for various architectures, where we start with the simplest model, which is for x86. Although the ISA is called x86 (Intel Corporation, 2025) its widely-accepted memory model is called x86-TSO (Owens et al., 2009)–TSO stands for *total store order*. The behavior of x86-TSO can operationally be described with local store buffers, whose values other threads cannot observe until they propagate to memory (which we explained in Section 1.1).

An operational semantics (Plotkin, 1981) defines the steps taken by the machine – often an abstract variant of it. For instance, the operational model of x86-TSO (Owens et al., 2009) explicitly captures the machine's internal state with store buffers in every CPU. With that representation, the machine's steps can precisely be described. However, we work with *axiomatic* models, where we do not model those architectural characteristics. Fortunately, the same paper introduces an equivalent axiomatic model, which captures the same behavior, but is more convenient within our context.

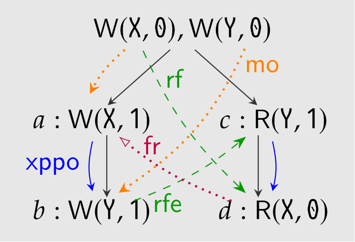
The primary consequence of store buffers is that R–R, R–W, and W–W pairs are ordered along po. In contrast, W–R pairs are not generally ordered, as we have seen above with the SB-weak-formal execution of the Store-Buffering program. The corresponding *x86 global happens before* axiom is as follows:

$$(\mathsf{implied} \cup \mathsf{xppo} \cup \mathsf{rfe} \cup \mathsf{fr} \cup \mathsf{mo}) \text{ is acyclic} \qquad \text{(XHB)}$$

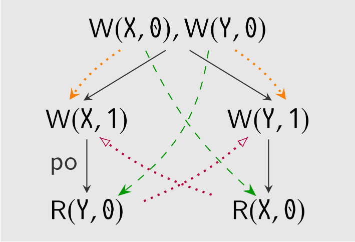
$$\begin{aligned}
\text{where } \mathsf{xppo} &\triangleq ((\mathsf{W}\times\mathsf{W}) \cup (\mathsf{R}\times\mathsf{W}) \cup (\mathsf{R}\times\mathsf{R})) \cap \mathsf{po}\\
\mathsf{implied} &\triangleq \mathsf{po};[At \cup \mathsf{F}] \cup [At \cup \mathsf{F}];\mathsf{po}\\
At &\triangleq \mathsf{dom}(\mathsf{rmw}) \cup \mathsf{codom}(\mathsf{rmw})
\end{aligned}$$

At a glance, its structure resembles that of SC, where implied and xppo order thread-local events, while rfe, fr, and mo relations capture the order among inter-thread events. Though, fr and mo, also include the thread-local fri and moi edges. The xppo –meaning *x86 preserved program order*– orders only the R–R, R–W, and W–W pairs along po, as we expected. In some cases, we may want to order W−R pairs too, for which we can place an intermediate *fence*. The cases of implied describe that thread-local order by stating that anything orders *before* and *after* a fence event F. Finally, implied also orders events produced by RMW operations—`CMPXCHG` on x86—which are ordered with all preceding and succeeding events, making those operations order like a fence. In the related *At* definition, the 'dom' and 'codom' functions respectively denote the domain and codomain of the relation they're applied to – rmw in this case. That means, when $R(x, y)$ then $x \in \mathrm{dom}(R)$ and $y \in \mathrm{codom}(R)$ for all R, x, and y.

Example. To illustrate x86's XHB axiom, we again inspect the weak behaviors of the Message-Passing and Store-Buffering programs – the latter we repeat from above.



(MP-weak-formal)



(SB-weak-formal)

In the MP-weak-formal execution, the po edge in the left thread is between W−W events, while it is between R−R events in the right thread. Following the definition of xppo, we obtain those edges between the events in both threads. We thus observe a 'xppo;rfe;xppo;fr' cycle, traversing $a \xrightarrow{\text{xppo}} b \xrightarrow{\text{rfe}} c \xrightarrow{\text{xppo}} d \xrightarrow{\text{fr}} a$. Because XHB explicitly disallows that cycle, the weak behavior is *impossible* on x86.

In contrast, the SB-weak-formal execution, the po edges in both threads are between W−R events. Those pairs are explicitly *excluded* from the definition of xppo. Hence, for that execution, there is *no* such cycle. As the XHB axiom disallows cycles –and there is no cycle here– the axiom is *satisfied* meaning the execution is allowed under the x86-TSO model and thus on the x86 architecture.

Note that in the x86-TSO model *all three* axioms Coherence, Atomicity, and XHB must be satisfied. Although we only inspected the final axiom in the examples for presentational simplicity, crucially, SB-weak-formal is only possible when it satisfies all three. That is the case here, as it also satisfies Coherence and Atomicity.

1.3.3 *Arm Semantics*

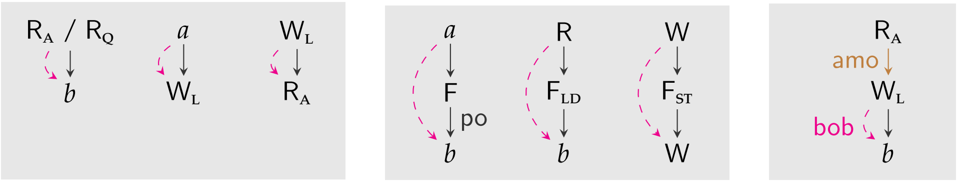
The Arm model (Pulte et al., 2017; Alglave et al., 2014, 2021) is more complex than x86-TSO, mostly because it is weaker, requiring a fine-grained specification of ordering constraints. Among the several variations of the Arm model, we present a variant of Alglave et al. (2021) adapted with the small fix[9] by Alglave (2022) –marked green– that is equivalent to the fix we will present in Chapter 3.

[9] The fix ensures Arm's $\texttt{CAS}_{\texttt{AL}}$ (*i.e.*, Arm's single-instruction RMW) orders like a full fence, like x86's `RMW`. We later explain that ordering issue in Section 3.3.3.

$$
\begin{aligned}
&\mathsf{ob} \text{ is irreflexive} && \text{(External)}\\
&\quad \text{where } \mathsf{ob} \triangleq (\mathsf{obs} \cup \mathsf{lob})^{+}\\
&\qquad\quad \mathsf{obs} \triangleq \mathsf{rfe} \cup \mathsf{moe} \cup \mathsf{fre}\\
&\qquad\quad \mathsf{lob} \triangleq (\mathsf{lws} \cup \mathsf{dob} \cup \mathsf{aob} \cup \mathsf{bob})^{+}\\
&\qquad\quad \mathsf{dob} \triangleq \mathsf{addr} \;\cup\; \mathsf{data} \;\cup\; \mathsf{ctrl};[\mathsf{W}] \;\cup\; (\mathsf{ctrl} \cup (\mathsf{addr};\mathsf{po}));[\mathsf{F}_{\mathrm{ISB}}];\mathsf{po};[\mathsf{R}]\\
&\qquad\qquad\quad \cup\; \mathsf{addr};\mathsf{po};[\mathsf{W}] \;\cup\; (\mathsf{ctrl} \cup \mathsf{data});\mathsf{lrs}\\
&\qquad\quad \mathsf{aob} \triangleq \mathsf{rmw} \;\cup\; [\mathsf{codom}(\mathsf{rmw})];\mathsf{rfi};[\mathsf{R}_{\mathrm{A}} \cup \mathsf{R}_{\mathrm{Q}}]\\
&\qquad\quad \mathsf{bob} \triangleq [\mathsf{R}_{\mathrm{A}} \cup \mathsf{R}_{\mathrm{Q}}];\mathsf{po} \;\cup\; \mathsf{po};[\mathsf{W}_{\mathrm{L}}] \;\cup\; [\mathsf{W}_{\mathrm{L}}];\mathsf{po}[\mathsf{R}_{\mathrm{A}}]\\
&\qquad\qquad\quad \cup\; \mathsf{po};[\mathsf{F}];\mathsf{po} \;\cup\; [\mathsf{R}];\mathsf{po};[\mathsf{F}_{\mathrm{LD}}];\mathsf{po} \;\cup\; [\mathsf{W}];\mathsf{po};[\mathsf{F}_{\mathrm{ST}}];\mathsf{po};[\mathsf{W}]\\
&\qquad\qquad\quad \cup\; [\mathsf{codom}([\mathsf{R}_{\mathrm{A}}];\mathsf{amo};[\mathsf{W}_{\mathrm{L}}])];\mathsf{po}\\
&\qquad\quad \mathsf{lrs} \;\triangleq [\mathsf{W}];(\mathsf{po}_{\mathsf{loc}} \setminus \mathsf{intervening\text{-}write}(\mathsf{po}_{\mathsf{loc}}));[\mathsf{R}]\\
&\qquad\quad \mathsf{lws} \triangleq \mathsf{po}_{\mathsf{loc}};[\mathsf{W}]
\end{aligned}
$$

Here, '+' is the transitive closure, meaning $R^{+}(x,y) \triangleq R(x,y) \vee (\exists z. R(x,z) \wedge R^{+}(z,y))$. Like the x86 XHB constraint, we see a clear distinction between thread-local and inter-thread relations; The ob (ordered-before) relation captures a full path, consisting of inter-thread obs (observed-by) and thread-local lob (locally-ordered-before) edges. Most complexity resides in the latter, as it captures many subtle ordering constraints, which are:

- lws (local write successor) – This states that Arm preserves the order between R–W and W–W pairs *to the same location* along po; but *not* R–R and W–R pairs.
- dob (dependency-ordered-before) – Dependencies capture some causality between po-related events that is otherwise absent from the execution graph. For instance, a program '`a = X; Y = a;`' reads a value from `X` and writes it to `Y`, resulting in an execution with two po-related events R(`X`, `42`) and W(`Y`, `42`), assuming that `X` contained `42` before. As those events should not execute out-of-order within a thread, Arm's semantics capture it with a data dependency between them.



R_{A} / R_{Q}
b
a
W_{L}
W_{L}
R_{A}
a
F
po
b
R
F_{LD}
b
W
F_{ST}
W
R_{A}
amo
W_{L}
bob
b

(a) bob for `LDAR` (R_{A}), `LDAPR` (R_{Q}), and `STLR` (W_{L})

(b) bob for fences

(c) bob after $\mathtt{CAS}_{\mathtt{AL}}$

Figure 4: Depiction of Arm's bob relation

The addr, data, and ctrl dependencies each preserve various such orders within a thread. All three are *primitive* relations for an Arm execution. However, we *do not* rely on Arm's dependencies in this dissertation and we will not discuss them in detail; Pulte et al. (2017) cover dependencies in more detail. The dob-contained lrs (local read successor) orders each R event after the po-preceding W event to the same location, which Alglave et al. (2021) explain in greater detail.

- aob (atomic-ordered-before) – This states how RMW instructions, regardless of their memory annotation, order within a thread. In particular, the rmw case states that an RMW–R event orders before the corresponding RMW–W event.

- bob (barrier-ordered-before) – This relations captures the ordering rules of fences and special read and write instructions, which Figure 4 also depicts. Arm's regular load LDR and store STR instructions are not generally ordered within a thread. However, Arm provides special concurrency primitives that do enforce an order:
  - LDAR generates a R_A event, which orders with po-successors.
  - STLR generates a W_L event, which orders with po-predecessors. Additionally, it orders before R_A events.
  - LDAPR generates a R_Q event, which orders with po-subsequent events, like R_A. However, unlike R_A, R_Q *does not* order after W_L events.

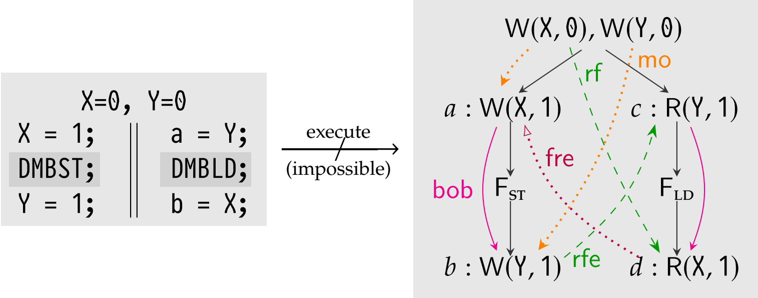
  Arm provides several fences, alternatively called "Data Memory Barriers":
  - DMBFF generates a F event, which orders *all* predecessors with *all* successors.
  - DMBLD generates a F_{LD} event, which orders preceding loads with all successors.
  - DMBST generates a F_{ST} event, which orders stores across it.

  Finally, Arm has various RMW instructions. Originally, it only had ll/sc instruction pairs, implemented with LX (load-exclusive) and SX (store-exclusive) instructions, that together read-and-write atomically. Within an execution, they generate a lxsx $\subseteq$ rmw relation between their events. Recently, Arm gained a single-instruction CAS (Compare-and-Swap) instruction (Alglave et al., 2021), producing another amo $\subseteq$ rmw relation. These lxsx and amo are *primitive* relations in an Arm execution, together forming rmw. The CAS instruction can have one of several order annotations, affecting how it orders with surrounding events. Notably, the $\texttt{CAS}_{\texttt{AL}}$ (Compare-and-Swap Acquire-Release) orders with predecessors and successors, like a full fence.

Any Arm execution must satisfy this External axiom, *in addition to* the general Coherence and Atomicity axioms.

Example. With the Arm model, we again inspect the SB-weak-formal and MP-weak-formal executions, shown above for x86. For the former, x86 does not include the W–R pairs in its xppo definition, thus producing no cycles, meaning its model allows that weak behavior for the Store-Buffering program. Similarly, Arm does *not* include those pairs in its definition of lob either, producing no cycles, thus also allowing the behavior. Hence, the behavior is observable on x86 and Arm, which both models formally capture.

For the MP-weak-formal example, x86 includes the W–W and R–R pairs in its xppo relation. The resulting ‘xppo;rfe;xppo;fr’ cycle disallows the weak behavior on x86. In contrast, as Arm *does not* include those same events in its lob relation, a similar cycle *does not* appear. That means this behavior is observable on Arm, while it is *not* observable on x86. We could, however, explicitly order the instructions to disallow the weak behavior on Arm, too; For instance, by placing fences in Message-Passing:



```
    X=0, Y=0
X = 1;  ||  a = Y;
DMBST;  ||  DMBLD;
Y = 1;  ||  b = X;
```

(MP-weak-Arm)

We placed lightweight fences in both threads, which preserve fewer orders between instructions than full fences, but are sufficient for this program. The DMBST store fence orders the W–W pair on the left thread, while the DMBLD load fence orders the R–R pair on the right thread. When we look at the same weak execution as before, we see the fence events in-between the memory accesses on both threads. In particular, the store fence generates a F_{ST} event, producing a bob relation between the W–W pair on the left thread; While the load fences generates a F_{LD} event, producing a bob relation between the R–R pair on the right thread. As a consequence, we now see a ‘bob;rfe;bob;fre’ cycle, traversing through $a \xrightarrow{\text{bob}} b \xrightarrow{\text{rfe}} c \xrightarrow{\text{bob}} d \xrightarrow{\text{fre}} a$. That cycle implies –following the definition of obs– a ‘bob;obs;bob;obs’ cycle that is disallowed by Arm’s External axiom. Hence, this particular weak behavior is *not* possible on Arm. Placing those fences ensured the program cannot show the additional weak behavior on Arm, enforcing x86-Arm robustness[10].

▶ **Remark** (Armv8 model variations) **.** Throughout the remaining chapters we use various variations of the Armv8 memory model, differing subtly from the one presented here. In Chapter 2 we use the Arm model by Pulte et al. (2017), as that chapter’s research precedes the release of the model by Alglave et al. (2021). We build upon that later model in subsequent chapters. In Chapter 3, we propose our own fix to it, which we use there. In Chapter 4 we use an equivalently-fixed model by

[10] Proving x86-Arm robustness demands that *all* behaviors on Arm are also observable on x86. Although we showed only one behavior here, we could enumerate all Arm behaviors for this program and demonstrate they also appear on x86. Intuitively, that is what we check in Chapters 2 to 5.

the original authors (Alglave, 2022; Alglave and Maranget, 2025) because we require its newer features (*i.e.,* mixed-size accesses). Finally, in Chapter 6, we use a similar model as in Chapter 3 but with the smaller fix proposed by Alglave et al. (2021).

These variations are equivalent (but refactored), except for *(i)* added CAS semantics (in Chapters 3 to 6) and *(ii)* added mixed-size semantics (in Chapter 4). In Section 3.3.3 we explain Arm's $\texttt{CAS}_{\texttt{AL}}$ ordering issue; Section 3.5 explains our fix.

1.3.4 *RISC-V Semantics*

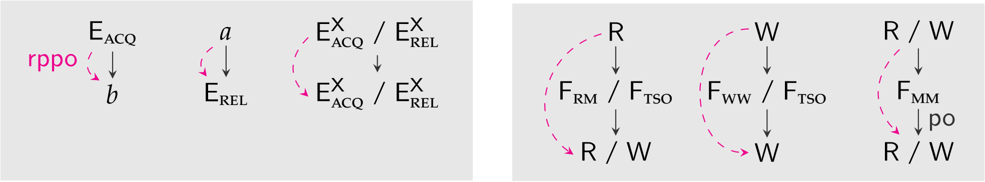
The RISC-V memory model (RISC-V International, 2024) is also weaker than x86's. Like Arm, RISC-V has load-acquire and store-release instructions. However, RISC-V has more instructions that give greater control over their ordering. In particular, it supports *ten* distinct memory fences, each ordering different kinds of memory instructions across it, unlike Arm's *three* memory fences.

$$\begin{aligned}
&(\mathsf{rppo} \cup \mathsf{mo} \cup \mathsf{rfe} \cup \mathsf{fr}) \text{ is acyclic} \qquad\qquad \text{(Model)}\\
&\quad\text{where}\\
&\quad \mathsf{rppo} \triangleq \mathsf{po_{loc}};[\mathsf{W}] \cup ([\mathsf{R}];\mathsf{poloc\text{-}no\text{-}w};[\mathsf{R}])\backslash\mathsf{rsw} \cup [\mathsf{W^X}];\mathsf{rfi}\\
&\qquad\qquad \cup\ \mathsf{fence} \cup [\mathsf{E_{ACQ}}];\mathsf{po} \cup \mathsf{po};[\mathsf{E_{REL}}] \cup [\mathsf{E^X_{ACQ}} \cup \mathsf{E^X_{REL}}];\mathsf{po};[\mathsf{E^X_{ACQ}} \cup \mathsf{E^X_{REL}}] \cup \mathsf{rmw}\\
&\qquad\qquad \cup\ \mathsf{addr} \cup \mathsf{data} \cup \mathsf{ctrl};[\mathsf{W}]\\
&\qquad\qquad \cup\ (\mathsf{addr} \cup \mathsf{data});\mathsf{rfi} \cup \mathsf{addr};\mathsf{po};[\mathsf{W}]\\
&\quad \mathsf{poloc\text{-}no\text{-}w} \triangleq \mathsf{po_{loc}} \setminus (\mathsf{po_{loc}}^{?};[\mathsf{W}];\mathsf{po_{loc}})\\
&\quad \mathsf{rsw} \qquad\quad\;\; \triangleq \mathsf{rf}^{-1};\mathsf{rf}\\
&\quad \mathsf{fence} \triangleq [\mathsf{R}];\mathsf{po};[\mathsf{F_{RR}}];\mathsf{po};[\mathsf{R}] \qquad \cup [\mathsf{R}];\mathsf{po};[\mathsf{F_{RW}}];\mathsf{po};[\mathsf{W}]\\
&\qquad\quad \cup [\mathsf{R}];\mathsf{po};[\mathsf{F_{RM}}];\mathsf{po};[\mathsf{R} \cup \mathsf{W}] \cup [\mathsf{W}];\mathsf{po};[\mathsf{F_{WR}}];\mathsf{po};[\mathsf{R}]\\
&\qquad\quad \cup [\mathsf{W}];\mathsf{po};[\mathsf{F_{WW}}];\mathsf{po};[\mathsf{W}] \quad \cup [\mathsf{W}];\mathsf{po};[\mathsf{F_{WR}}];\mathsf{po};[\mathsf{R}]\\
&\qquad\quad \cup [\mathsf{R} \cup \mathsf{W}];\mathsf{po};[\mathsf{F_{MR}}];\mathsf{po};[\mathsf{R}] \cup [\mathsf{R} \cup \mathsf{W}];\mathsf{po};[\mathsf{F_{MW}}];\mathsf{po};[\mathsf{W}]\\
&\qquad\quad \cup [\mathsf{R} \cup \mathsf{W}];\mathsf{po};[\mathsf{F_{MM}}];\mathsf{po};[\mathsf{R} \cup \mathsf{W}]\\
&\qquad\quad \cup [\mathsf{W}];\mathsf{po};[\mathsf{F_{TSO}}];\mathsf{po};[\mathsf{W}] \quad \cup [\mathsf{R}];\mathsf{po};[\mathsf{F_{TSO}}];\mathsf{po};[\mathsf{R} \cup \mathsf{W}]
\end{aligned}$$

Here, $\mathsf{E^X}$ denotes an event generated by a special RMW, *load reserved* (LR), or *store-conditional* (SC[11]) instruction. Again, there is a clear distinction between thread-local edges in rppo[12] and inter-thread edges mo, rfe, and fr. Like for x86-TSO, the latter also includes thread-local moi and fri relations. We explain the four categories in rppo below, each corresponding to one of the four lines in the definition above, in order.

[11] Confusingly, both "Sequential Consistency" and "Store Conditional" conventionally abbreviate to "SC". Throughout this dissertation, we use a monospace font for the latter RISC-V instruction: SC, but will usually refrain from these abbreviations.

[12] We denote RISC-V's *preserved program order* with rppo to avoid ambiguity with x86's xppo.



(a) rppo for acquire/release instructions

(b) rppo for fences. Note that F_{TSO} includes the order of both F_{RM} and F_{WW}.

Figure 5: Depiction of several cases of RISC-V's rppo relation

- *Overlapping-Address Orderings* – The first three cases capture same-location po-related events. First, a write orders after any same-location R or W event. Second, two same-location R events are ordered when *(i)* there is no intermediate same-location write (captured by poloc-no-w) *and (ii)* the two R events read from (rf) different W events. The '?' in poloc-no-w denotes a *reflexive closure*, meaning $R^?(x,y) \triangleq R(x,y) \vee x \equiv y$. Third, a R orders after a same-location W^X that it reads from (rf).

  RISC-V's model separately considers events generated by special `RMW`, `LR`, and `SC` instructions. A successful `RMW` generates rmw-related R^X and W^X events, while a failing `RMW` generates only a R^X. A *load-reserved* (`LR`) instruction and `RMW` generate a R^X event. `LR` and `SC` together operate as an atomic load-linked store-conditional (ll/sc) read-modify-write, generating a rmw in the execution (like Arm's `LX` and `SX`). We denote the combination with R^X,$W^X \subseteq E^X$.

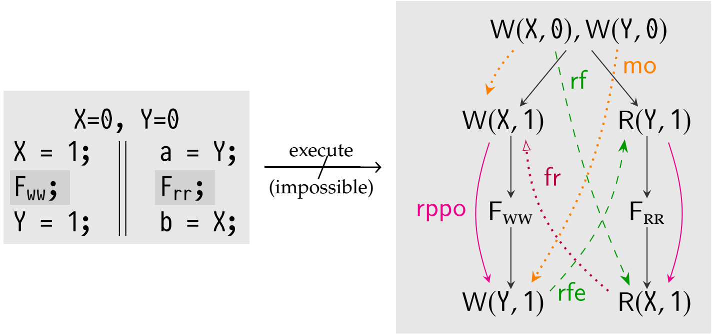
- *Explicit Synchronization* – Some instructions are explicitly ordered by fences or acquire/release annotations, like in Arm – we depict some in Figure 5. In particular, RISC-V has numerous fences (included in fence), each of which orders different instructions across it. For instance, $F_{\mathtt{rm}}$ (generating F_{RM}) orders like Arm's `DMBLD`, $F_{\mathtt{ww}}$ (generating F_{WW}) orders like Arm's `DMBST`, and $F_{\mathtt{mm}}$ (generating F_{MM}) orders like Arm's `DMBFF`. However, RISC-V additionally includes, for instance, a $F_{\mathtt{wr}}$ instruction (generating F_{WR}), ordering preceding writes with succeeding reads; or a $F_{\mathtt{tso}}$ that combines the ordering constraints of $F_{\mathtt{rm}}$ and $F_{\mathtt{ww}}$, thus capturing the default orders of x86-TSO.

  In RISC-V, instructions can also have an acquire or release annotation, officially following *release consistency* (Gharachorloo et al., 1990; RISC-V International, 2024). Notably, it orders its special `RMW`, load-reserved, and store-conditional instructions *with each other*, provided that both instructions have an acquire or release annotation (or both). For instance, a store-conditional with release annotation orders *before* a load-reserved with acquire annotation. Although RISC-V has multiple single-instruction RMWs, we primarily use $\mathtt{RMW}_{\mathtt{AL}}$ throughout this dissertation, which has both acquire and release annotations.

- *Syntactic Dependencies* and *Pipeline Dependencies* – The syntactic and pipeline dependencies are two separate categories which both capture dependencies. Like Arm, RISC-V has the *primitive* relations addr, data, and ctrl. As we do not rely upon dependencies in the remaining chapters, we will not explain them here further.

Any RISC-V execution must satisfy this Model axiom, *in addition to* the general Coherence and Atomicity axioms.

Example. With this RISC-V model, we again inspect the SB-weak-formal and MP-weak-formal executions. Similar, to Arm, it does not order W–W and R–R pairs in the former, nor the W–R pairs in the latter. Formally, that is because RISC-V does not include those pairs in its rppo definition. However, to enforce *x86–RISC-V* robustness, we must ensure the Message-Passing program cannot show the MP-weak behavior on RISC-V – as it is also impossible on x86. As with Arm, we can again place fences to ensure robustness.



```
     X=0, Y=0
X = 1;  ||  a = Y;
Fww;    ||  Frr;
Y = 1;  ||  b = X;
```

(MP-weak-RISC-V)

We see that the $\mathsf{F_{ww}}$ generates the rppo edge in the left thread between the W–W pair. The $\mathsf{F_{rr}}$ generates the rppo edge in the right thread between the R–R pair. Between the event in the thread, we thus obtain a 'rppo;rfe;rppo;fr' cycle, which is disallowed by the RISC-V Model constraint, making this weak behavior impossible on RISC-V.

Particularly observe that we judiciously placed RISC-V's lightweight fences. Although the $\mathsf{F_{ww}}$ orders like Arm's DMBST, the $\mathsf{F_{rr}}$ orders weaker than Arm's DMBLD because the $\mathsf{F_{rr}}$ does not order R–W pairs. However, as this particular program does not include such pairs, the $\mathsf{F_{rr}}$ fence is sufficient to prevent the weak behavior.

1.3.5 *Overview of Architecture Primitives*

We give an overview of the concurrency primitives in architectures with the events they generate in Figure 6. Although the terminology of instructions overlaps between architectures, instructions do not correspond one-to-one. For instance, x86's regular load (RMOV – R) orders *stronger than* Arm's regular load (LDR – R) but similar to Arm's load-acquirePC (LDAPR – $\mathsf{R_Q}$). RISC-V's load-acquire ($\mathsf{LD_{acq}}$ – $\mathsf{R_{ACQ}}$) orders like Arm's load-acquirePC. The table highlights only the syntactic similarities between architectures, but only their formal models –as we have defined above– precisely capture their semantic differences.

Instruction Type	x86		Arm		RISC-V	
	instr.	event	instr.	event	instr.	event
Load	RMOV	R	LDR	R	LD	R
Load-acquire			LDAR	R_{A}	$\mathtt{LD}_{\mathtt{acq}}$	R_{ACQ}
Load-acquirePC			LDAPR	R_{Q}		
Store	WMOV	W	STR	W	ST	W
Store-release			STLR	W_{L}	$\mathtt{ST}_{\mathtt{rel}}$	W_{REL}
Full fence	MFENCE	F	DMBFF	F	$\mathtt{F}_{\mathtt{mm}}$	F_{MM}
Load fence			DMBLD	F_{LD}	$\mathtt{F}_{\mathtt{rm}}$	F_{RM}
Store fence			DMBST	F_{ST}	$\mathtt{F}_{\mathtt{ww}}$	F_{WW}
RMW	CMPXCHG	rmw	$\mathtt{CAS}_{\mathtt{AL}}$	amo_{AL}	$\mathtt{RMW}_{\mathtt{AL}}$	rmw_{AL}
ll/sc			LX/SX	lxsx	LR/SC	rmw

$$\mathsf{amo}_{AL} \triangleq [R_{A}];\mathsf{amo};[W_{L}], \quad \mathsf{rmw}_{AL} \triangleq [R_{AL}];\mathsf{rmw};[W_{AL}]$$

Figure 6: Concurrency primitives in architectures with their generated events and relations; Not exhaustive, but covers all used throughout this dissertation.

1.4 HIGH-LEVEL THEOREM STATEMENT

To ensure that any weak memory transformation is correct, we must ensure it introduces no additional behavior. In the previous subsections we have seen that programs may non-deterministically produce one of multiple execution graphs during execution. For instance, when translating the Message-Passing program from x86 to Arm, we take care not to introduce the MP-weak behavior. We visualize that correctness criterion below:

$$\begin{tikzcd}
\mathbb{P}^{s} \arrow[r, "\text{map}"] \arrow[d, "\text{execute}"'] & \mathbb{P}^{t} \arrow[d, "\text{execute}"] \\
[\![\mathbb{P}^{s}]\!]_{M^{s}} \arrow[r, phantom, "\supseteq"] & [\![\mathbb{P}^{t}]\!]_{M^{t}}
\end{tikzcd}$$

The source program $\mathbb{P}^{s}$ executes under the source memory model M^{s}, producing *a set* of execution graphs, denoted with $[\![\mathbb{P}^{s}]\!]_{M^{s}}$. The 'map' function transforms the syntax of the source program $\mathbb{P}^{s}$ into the target program $\mathbb{P}^{t}$. That target program executes under the target memory model M^{t}, also producing *a set* of execution graphs, denoted with $[\![\mathbb{P}^{t}]\!]_{M^{t}}$. As the transformation *must not* introduce new executions, we strictly require that $[\![\mathbb{P}^{t}]\!]_{M^{t}} \subseteq [\![\mathbb{P}^{s}]\!]_{M^{s}}$. In other words, any executions shown by $\mathbb{P}^{t}$ must also have been shown by $\mathbb{P}^{s}$. We formally state that as:

▶ **Definition 3** (Correct Transformation). Given mapping scheme that generates a target program $\mathbb{P}^t$ from the source program $\mathbb{P}^s$ (for any $\mathbb{P}^s$), that scheme is correct if for each M^t-consistent execution $\mathsf{X}^t \in [\![\mathbb{P}^t]\!]_{\mathsf{M}^t}$ there exists a M^s-consistent execution $\mathsf{X}^s \in [\![\mathbb{P}^s]\!]_{\mathsf{M}^s}$ such that $\mathsf{Behavior}(\mathsf{X}^t) = \mathsf{Behavior}(\mathsf{X}^s)$.

Correctness (Definition 3) applies to any transformation we consider, not only the x86-to-Arm mappings. For instance, it also applies to optimizations (*e.g.*, in Chapters 2 and 3), where both $\mathbb{P}^s$ and $\mathbb{P}^t$ execute in the same memory model. Those optimization must not introduce additional behaviors either. Other verification techniques should use the same correctness criterion. We do that in Chapter 6, where we check whether an execution observed at runtime satisfies this same theorem.

Precise Mappings. Although correctness is crucial, architecture mappings should also be precise. Consider, for instance, the naive scheme inserting full fences in-between any two adjacent memory accesses; That scheme is correct–meaning it satisfies Definition 3–but enforces too much orders, harming performance. Hence, in addition to correctness, mappings schemes must be *precise*. Intuitively, a precise mapping scheme inserts only as many fences as necessary. More formally, we state it as:

▶ **Definition 4** (Precise Transformation). A correct mapping scheme is precise if for each fence in the mapping, there exists a program where the fence is necessary and sufficient to preserve correctness, *i.e.*, no weaker fence is sufficient and no stronger fence is necessary.

In contrast to correctness, which requires elaborate proofs over the axiomatic semantics, precision can often be demonstrated by example. We must only show, independently for every fence in the mapping, that there exists a program satisfying: *(i)* with the fence it is correct and *(ii)* without the fence it is incorrect; That also requires demonstrating no weaker fence exists for which the mapping is correct. For instance, the fences inserted in the program producing MP-weak-Arm are precise. Independently, removing either the `DMBST` on the first thread or the `DMBLD` on the second thread would break the cycle, erroneously allowing the execution. As Arm has no weaker fences that could substitute either, those inserted fences are precise.

NOTATIONS

Figure 7 contains an overview of the notations we use throughout this dissertation for binary relations.

Name	Symbol	Definition
Composition	;	$(R_1;R_2)(x,y) \triangleq \exists z.R_1(x,z) \land R_2(z,y)$
Identity Relation	[_]	$[S](x,y) \triangleq S(x,x) \land x{\equiv}y$
Inverse	$^{-1}$	$R^{-1}(x,y) \triangleq R(y,x)$
Transitive Closure	$^{+}$	$R^{+}(x,y) \triangleq R(x,y) \lor (\exists z.R(x,z){\land}R^{+}(z,y))$
Reflexive Closure	$^{?}$	$R^{?}(x,y) \triangleq R(x,y) \lor x{\equiv}y$
Immediate	$_{\mathsf{imm}}$	$R_{\mathsf{imm}}(x,y) \triangleq R(x,y) \land \neg\exists z.(R(x,z) \land R(z,y))$
Same-Location	R_{loc}	$R_{\mathsf{loc}}(x,y) \triangleq R(x,y) \land x.\mathsf{loc}{\equiv}y.\mathsf{loc}$

Figure 7: Overview of notations for binary relations

Chapter 2

Static Program Translation with Mapping Schemes

ABSTRACT In this chapter, we propose LASAGNE, an end-to-end static binary translator with precise translation rules between x86 and Arm weak memory concurrency semantics. First, we propose a concurrency model for LASAGNE's intermediate representation and formally proved mappings between the IR and the two architectures. The memory ordering is preserved by adding fences in the translated code. Finally, we prove several common transformations correct on that IR, which LASAGNE uses to optimize programs while translating them from x86 to Arm. In practice, we observe our optimizations significantly reduce the number of fences and their runtime overhead.

2.1 INTRODUCTION

Static Binary Translation (SBT) is a process for automatically rewriting, ahead-of-time, a program's machine code from the original architecture to a target architecture. Because SBT works on the machine code itself, access to the original source-code is not required. Crucially, the translation must preserve the semantics of the original binary, as specified by the original architecture, while also optimizing the target binary. Although SBT tools have gained popularity (Guo et al., 2016; Fu et al., 2018), their support of several advanced architectural features is often limited.

Furthermore, these SBT tools also cannot translate concurrent binaries (Yadavalli and Smith, 2019; Trail of Bits, 2022; Avast Developers, 2022). This is because of the mismatches of the *weak memory consistency model* in different architectures, which governs the valid orderings of memory accesses. To address this problem, the translation tools must reason about the consistency models for *correct* and *efficient* translation.

In this chapter, we address the challenge of developing efficient translation between x86 and Arm concurrency semantics through LLVM's IR. The x86 and Arm architectures have different *memory ordering* semantics, which results in different *memory ordering* rules. Therefore, we need a concurrency model for the LLVM primitives that enables precise mapping schemes between LLVM and these architectures, while also allowing code transformations. We note that existing concurrency models (Chakraborty and Vafeiadis, 2017; Lahav et al., 2017; Kang et al., 2017; Chakraborty and Vafeiadis, 2019; Podkopaev et al., 2019), which were originally developed for compilation of high-level languages, do not suffice to satisfy all these requirements. Hence, to bridge this gap, we propose LIMM (LLVM IR Memory Model). We use this model to design precise mapping schemes and prove them correct. We implement these mapping schemes in LASAGNE, an end-to-end static binary translator.

Our primary contribution is the *LLVM IR Memory Model*, named LIMM (Section 2.5). Based on LIMM, we design mapping schemes and transformations, which we prove correct in the Agda proof assistant (Agda Team, 2025a) (Section 2.6). These transformations –*e.g.,* instruction elimination and reordering– are commonly used in existing LLVM optimization passes. We implement our mappings in Lasagne (Section 2.7).

2.2 BACKGROUND

Our approach is based on static binary translation. Similar to modern compilers, the architecture of modern static binary translators have a 3-phases structure. In both cases, their first phase (the compiler frontend and binary lifter) translate the input program to an IR, *e.g.,* LLVM IR. This IR code is then optimized and finally compiled down to its final binary format for a given architecture. There are two key benefits to this approach: First, the lifted code can be re-targeted to multiple architectures. Second, existing optimizations can directly be used on the lifted code.

State-of-the-Art Binary Lifters. Lifting the source binary requires correctly mapping source to target instructions, discover global values, and reconstruct the control flow graph. Several state-of-the-art binary lifters target an intermediate representation, *e.g.,* LLVM IR, to ease this process (Bougacha, 2022; Trail of Bits, 2022; Yadavalli and Smith, 2019; Avast Developers, 2022; Shen et al., 2012a; Bellard, 2005; Spink et al., 2019; Hong et al., 2012; Cota et al., 2017). The existing lifting tools primarily target sequential programs and do not handle concurrency, meaning they ignore the differences in memory consistency models altogether.

2.3 MOTIVATION

It is well-known that any transformation (*i.e.,* mapping or optimization) written for sequential programs may not be correct for concurrent programs (Ševčík, 2011; Morisset et al., 2013; Vafeiadis et al., 2015; Chakraborty and Vafeiadis, 2016). As state-of-the-art SBT tools are written for sequential programs (Bougacha, 2022; Trail of Bits, 2022; Yadavalli and Smith, 2019; Avast Developers, 2022; Shen et al., 2012a), using them to translate concurrent programs may lead to erroneous program behavior.

As a concrete example, consider the translation in Figure 8. mctoll (Yadavalli and Smith, 2019) lifts the x86 program in Figure 8a to the LLVM IR in Figure 8b, where it translates the shared variable accesses in x86 to non-atomic accesses. Next, LLVM reorders the shared memory non-atomic accesses (na) and generates the optimized IR in Figure 8c. Finally, LLVM generates the Arm program in Figure 8d that may exhibit program outcomes that were originally not allowed in x86. The error results from the lack of reasoning about concurrency at the IR level. To do so, the IR needs a concurrency model. Thus, the combination of mctoll and LLVM raises a question: *What is the concurrency model of the IR*?

To translate concurrent programs correctly and efficiently, we need a formal model before we can formally phrase those criteria. That formal concurrency model for the IR must fulfill the following desired properties:

- **Precise mapping schemes.** The concurrency model must facilitate precise mapping schemes from the source to the IR and from the IR to the target. In particular, translating through the IR should *not* add more fences than when directly translating from x86 to Arm. In addition, that overall translation from x86 to Arm should not add more fences than necessary to preserve the original x86 program's memory orders – to avoid harming performance.

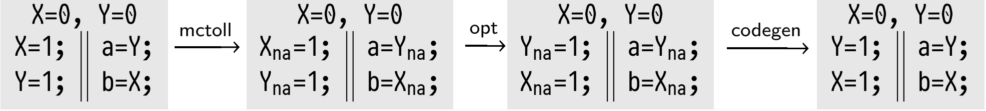
- **Optimized.** The IR should allow common transformations, including shared memory access reordering, elimination, and redundant fence elimination. Proving these transformation correct ensures LLVM can safely apply common optimizations.

2.4 OVERVIEW

A key aspect of our static translator Lasagne is strong-to-weak binary translation by strategically placing memory fences to correctly emulate the memory ordering behavior of the source architecture (*i.e.,* x86) on the target architecture (*i.e.,* Arm). Our overarching goal is to support *correct* and *optimized* placement of fences, so that we emulate the source architecture faithfully, without introducing run-time overheads.

#1: Binary Lifting. First we lift x86 binaries into LLVM bitcode. The main challenge in binary lifting comes from reconstructing, from the machine code, higher-level abstractions that have been lost in the compilation process. While lifting the binary to LLVM IR, it is important to identify these abstractions to enable more aggressive optimizations in the subsequent stages.

#2: LIMM: IR Concurrency Model. We introduce LIMM (LLVM IR Memory Model), which acts as Lasagne's formal concurrency model. LIMM extends the concurrency primitives in the LLVM IR. The semantics of LLVM non-atomic accesses differ from their corresponding x86 and Arm load and store accesses. On x86, the order of



(a) x86 (b) Unoptimized LLVM IR (c) Optimized LLVM IR (d) Arm

Figure 8: Example of incorrect x86 to Arm translation by mctoll + LLVM. Suffix na denotes the non-atomic accesses in LLVM IR. Outcome $a = 1, b = 0$ is disallowed in the x86 program but allowed in the generated Arm program.

RMOV-RMOV, RMOV-WMOV, and WMOV-WMOV access pairs is preserved, whereas on LLVM non-atomic load and store accesses are always unordered (on different locations). The Arm concurrency model disallows the removal of false dependencies (Pulte et al., 2017) as these dependencies enforce certain orders between memory accesses. In contrast, LLVM regularly removes false dependencies in various optimizations. To allow these optimizations, LIMM does not order any accesses based on dependencies. We describe LIMM in Section 2.5.

#3: Translation Correctness in LIMM. Based on LIMM, we define precise mapping schemes for translating between architectures, and reason about the correctness of the common transformations on LIMM. More specifically, we identify the safe/unsafe reordering of independent shared memory accesses and fences. We also identify safe elimination of redundant shared memory accesses. The main challenge is to formally prove the correctness of the mapping schemes and the safe transformations. We discuss the mapping schemes and the transformations in Section 2.6.

#4: Implementing LIMM Translations. We implement our mapping schemes in Lasagne, which inserts fences into the lifted LLVM IR. In particular, these schemes insert leading or trailing fences for shared accesses in programs lifted from x86 to LLVM IR. In Section 2.6, we describe reordering and elimination transformations, which LLVM regularly performs; these remain correct under LIMM. We discuss further details in Section 2.7.

2.5 LIMM WEAK MEMORY MODEL

In this section, we describe our LIMM formal weak memory model, which we later use to prove Lasagne's mapping of weak memory primitives from x86 to Arm and to prove optimizations on LIMM correct. We use the axiomatic weak memory concurrency models as presented Chapter 1.

Syntax. As Lasagne builds on LLVM, we define the formal model over concurrency primitives in LLVM IR. In particular, we use LLVM's non-atomic load ($\mathtt{ld}_{\mathtt{na}}$) and store ($\mathtt{st}_{\mathtt{na}}$) instructions, and atomic $\mathtt{RMW}_{\mathtt{sc}}$ accesses (*i.e.,* RMW with seq_cst order annotation). To order non-atomic instructions, we use various LLVM fences. $\mathtt{F}_{\mathtt{sc}}$ (*i.e.,* fence with seq_cst order annotation) is a *full* fence, like MFENCE in x86 and DMBFF in Arm. We add $\mathtt{F}_{\mathtt{rm}}$ and $\mathtt{F}_{\mathtt{ww}}$ instructions to LLVM IR, which are similar to the DMBLD and DMBST fences in Arm, respectively. An $\mathtt{F}_{\mathtt{rm}}$ orders preceding loads with any succeeding memory accesses – M refers to any memory access. Any write-write pair is ordered by an intermediate $\mathtt{F}_{\mathtt{ww}}$ fence.

$$\begin{array}{lll}
\mathsf{ghb} \text{ is irreflexive where} & & (\mathsf{GOrd}) \\
\mathsf{ghb} \triangleq (\mathsf{ord} \cup \mathsf{rfe} \cup \mathsf{moe} \cup \mathsf{fre})^{+} \text{ where} & & \\
\mathsf{ord} \triangleq [\mathsf{R}];\mathsf{po};[\mathsf{F}_{\text{RM}}];\mathsf{po};[\mathsf{R} \cup \mathsf{W}] & & (\mathsf{ord}_1) \\
\quad \cup\, [\mathsf{W}];\mathsf{po};[\mathsf{F}_{\text{WW}}];\mathsf{po};[\mathsf{W}] & & (\mathsf{ord}_2) \\
\quad \cup\, [\mathsf{F}_{\text{SC}} \cup \mathsf{R}_{\text{SC}} \cup \mathsf{codom}(\mathsf{rmw})];\mathsf{po} & & (\mathsf{ord}_3) \\
\quad \cup\, \mathsf{po};[\mathsf{F}_{\text{SC}} \cup \mathsf{W}_{\text{SC}} \cup \mathsf{dom}(\mathsf{rmw})] & & (\mathsf{ord}_4)
\end{array}$$

Figure 9: LIMM Concurrency Model. Coherence and Atomicity also hold in LIMM.

Events. A LIMM program generates the following events upon executing:

- For the non-atomic load ($\mathtt{ld}_{\mathtt{na}}$) and store ($\mathtt{st}_{\mathtt{na}}$) accesses, we generate $\mathsf{R}_{\text{NA}}(x, v)$ and $\mathsf{W}_{\text{NA}}(x, v)$ events, respectively.
- A successful $\mathtt{RMW}_{\mathtt{sc}}(x, v_r, v_w)$ generates a pair of $\mathsf{R}_{\text{SC}}(x, v_r)$ and $\mathsf{W}_{\text{SC}}(x, v_w)$ events which are rmw-related. If it reads v' and fails, it generates a single $\mathsf{R}_{\text{SC}}(x, v')$.
- The $\mathtt{F}_{\mathtt{sc}}$, $\mathtt{F}_{\mathtt{rm}}$, and $\mathtt{F}_{\mathtt{ww}}$ fences generate fence events F_{SC}, F_{RM}, and F_{WW}, respectively.

Finally, in LIMM, $\mathsf{R} = \mathsf{R}_{\text{NA}} \cup \mathsf{R}_{\text{SC}}$ and $\mathsf{W} = \mathsf{W}_{\text{NA}} \cup \mathsf{W}_{\text{SC}}$.

Relations. We define *order* (ord) and global-happen-before (ghb) relations in Figure 9. The ord relation relates thread-local po-related events, while rfe, moe, and fre capture inter-thread communication. A pair of po-related events (a, b) are in ord relation when:

(ord_1) There is an intermediate F_{RM} event, a is a read, and b is a memory access;

(ord_2) a and b are writes with an intermediate F_{WW} event;

(ord_3) a is an F_{SC} event or generated from a $\mathtt{RMW}_{\mathtt{sc}}$ (*i.e.*, R_{SC} or codom(rmw)); or

(ord_4) b is an F_{SC} event or generated from a successful $\mathtt{RMW}_{\mathtt{sc}}$ (*i.e.*, W_{SC} or dom(rmw)).

Note that we do not define any ordering based on dependencies in the IR. This is because LLVM may eliminate false dependencies. Such eliminations could introduce disallowed behavior, which would render the translations incorrect.

Axioms. We can now formally define a ghb relation on events across threads, which we show in Figure 9. In an execution graph, ghb(a, b) states that a path from a to b exists, consisting of ord and external relations rfe, moe, fre. In addition to the LIMM-specific GOrd axiom, consistency with LIMM requires the common Coherence and Atomicity (from Section 1.3).

2.6 MAPPING SCHEMES

We use LIMM from the previous section to correctly reason about concurrency in the IR. Our main objective is to define a precise mapping from x86 to Arm, which goes through the IR. Additionally, we reason about the IR-to-IR optimizing transformations on LIMM. We mechanize the correctness proofs in Agda (Agda Team, 2025a).

x86		IR
RMOV	→	$\mathtt{ld}_{\mathtt{na}};\mathtt{F}_{\mathtt{rm}}$
WMOV	→	$\mathtt{F}_{\mathtt{ww}};\mathtt{st}_{\mathtt{na}}$
CMPXCHG	→	$\mathtt{RMW}_{\mathtt{sc}}$
MFENCE	→	$\mathtt{F}_{\mathtt{sc}}$

(a) x86 to IR

IR		Arm
$\mathtt{ld}_{\mathtt{na}}$	→	LDR
$\mathtt{st}_{\mathtt{na}}$	→	STR
$\mathtt{RMW}_{\mathtt{sc}}$	→	DMBFF; RMW; DMBFF
$\mathtt{F}_{\mathtt{rm}}$	→	DMBLD
$\mathtt{F}_{\mathtt{ww}}$	→	DMBST
$\mathtt{F}_{\mathtt{sc}}$	→	DMBFF

(b) IR to Arm

x86		IR		Arm
RMOV	→	$\mathtt{ld}_{\mathtt{na}};\mathtt{F}_{\mathtt{rm}}$	→	LDR; DMBLD
WMOV	→	$\mathtt{F}_{\mathtt{ww}};\mathtt{st}_{\mathtt{na}}$	→	DMBST; STR
CMPXCHG	→	$\mathtt{RMW}_{\mathtt{sc}}$	→	DMBFF; RMW; DMBFF
MFENCE	→	$\mathtt{F}_{\mathtt{sc}}$	→	DMBFF

(c) x86 to IR to Arm (combining *(a)* and *(b)*)

Figure 10: Verified mappings from x86 to Arm through LASAGNE's IR.

2.6.1 *Mapping Correctness*

We propose the mapping schemes of the concurrency primitives between x86 and Arm through the primitives in the IR, shown in Figure 10. For each mapping scheme, we prove it correct (Definition 3 in Section 1.4) in Agda and show it is precise (Definition 4).

x86 to IR (Figure 10a). The load and store in the IR are weaker than those of x86. So we map an x86 load to an IR load with a *trailing* $\mathtt{F}_{\mathtt{rm}}$. The $\mathtt{F}_{\mathtt{rm}}$ orders preceding loads with any successor memory accesses. Similarly, an x86 store is mapped to an IR store with a *leading* $\mathtt{F}_{\mathtt{ww}}$. The $\mathtt{F}_{\mathtt{ww}}$ orders the store with any preceding store. An x86 RMW is

```
   X = X = 0;
X = 1; || a = Y;
Y = 1; || b = X;
```

→

```
       X = Y = 0;
X_na = 1; || a = Y_na;
F_ww;     || F_rm;
Y_na = 1; || b = X_na;
```

→

```
     X = Y = 0;
Y = 1;  || a = Y;
DMBST;  || DMBLD;
X = 1;  || b = X;
```

(a) x86 (b) IR (c) Arm

Figure 11: x86, IR, and Arm versions of the Message-Passing program in x86 to IR to Arm translations by the proposed mapping schemes in Figure 10.

mapped to an $\texttt{RMW}_{\texttt{sc}}$ in the IR – a successful atomic-update acts as a full fence in both x86 and the IR. Finally, an `MFENCE` maps to an $\texttt{F}_{\texttt{sc}}$ fence.

▶ **Theorem 1** (x86→IR Correct). The x86→IR mapping is correct (Definition 3).

We have proved Theorem 1 in Agda.

▶ **Theorem 2** (x86→IR Precise). The x86→IR mapping is precise (Definition 4).

To prove Theorem 2, we must demonstrate that the $\texttt{F}_{\texttt{rm}}$ and $\texttt{F}_{\texttt{ww}}$ are required. In Figure 11a and Figure 11b, we show the x86 program and the generated IR, respectively. The IR program disallows the outcome a=1,b=0, similar to the x86 program. Without either of those two fences, that outcome would be allowed for the IR program, making the mapping incorrect. In addition, the IR does not provide any weaker fences than $\texttt{F}_{\texttt{rm}}$ and $\texttt{F}_{\texttt{ww}}$. Hence, the x86→IR mapping scheme is precise.

IR to Arm (Figure 10b). We map an IR load to Arm `LDR` and IR store to Arm `STR` instruction. The IR $\texttt{RMW}_{\texttt{sc}}$ maps to an Arm `RMW` primitive, around which we insert leading and trailing `DMBFF` fences. Here, Arm's `RMW` operation refers to the load-linked store-conditional (ll/sc) instruction pair (*i.e.*, `LX` / `SX` instructions). As those do not enforce sufficient ordering by themselves, we surround them with the `DMBFF` full fences. Finally, we map $\texttt{F}_{\texttt{rm}}$, $\texttt{F}_{\texttt{ww}}$, $\texttt{F}_{\texttt{sc}}$ fences in the IR to `DMBLD`, `DMBST`, `DMBFF` accesses, respectively.

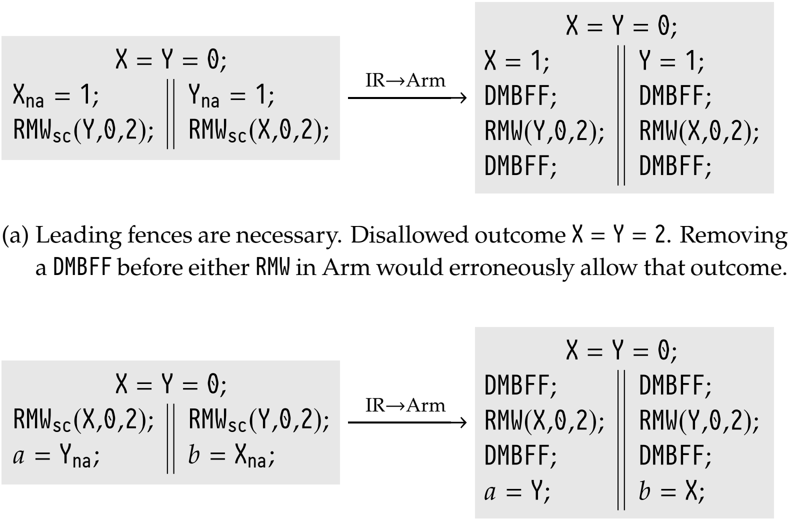
▶ **Theorem 3** (IR→Arm Correct). The IR→Arm mapping is correct (Definition 3).

We have proved Theorem 3 in Agda.

▶ **Theorem 4** (IR→Arm Precise). The IR→Arm mapping is precise (Definition 4).

To prove Theorem 4, we must demonstrate that all fences are required. For the `DMBLD` and `DMBST` fences surrounding the accesses, we use a similar argument as for the x86→IR mapping above. In Figures 11b and 11c, we show the IR program and the mapped Arm program, respectively. As the former disallows the outcome a=1,b=0, the latter should disallow it too. If we had removed either the `DMBST` or `DMBLD` fence from that program, that additional behavior would be observable, making the mapping incorrect. In addition, Arm does not have weaker fences than `DMBLD` and `DMBST` to preserve the same ordering.

Finally, we must demonstrate that the `DMBFF` instructions surrounding the `RMW` are necessary, which Figure 12 illustrates. Figure 12a shows that the `DMBFF` *before* `RMW` is necessary. Without that fence, a preceding W would be unordered with the `RMW`–R event. Figure 12b shows that the `DMBFF` *after* `RMW` is necessary. Without that fence, the `RMW`–W would be unordered with a succeeding R. In both cases, the `DMBFF` fence orders a W–R pair, for which no weaker fence exists in Arm – note that neither `DMBLD` nor `DMBST` order W–R pairs. Both `DMBFF` fences surrounding the `RMW` are thus necessary in Arm. As all fences in the IR→Arm mapping are necessary, the mapping is precise.



X = Y = 0;

X_{na} = 1;	Y_{na} = 1;
RMW_{sc}(Y,0,2);	RMW_{sc}(X,0,2);

X = Y = 0;

X = 1;	Y = 1;
DMBFF;	DMBFF;
RMW(Y,0,2);	RMW(X,0,2);
DMBFF;	DMBFF;

(a) Leading fences are necessary. Disallowed outcome X = Y = 2. Removing a DMBFF before either RMW in Arm would erroneously allow that outcome.

X = Y = 0;

RMW_{sc}(X,0,2);	RMW_{sc}(Y,0,2);
a = Y_{na};	b = X_{na};

X = Y = 0;

DMBFF;	DMBFF;
RMW(X,0,2);	RMW(Y,0,2);
DMBFF;	DMBFF;
a = Y;	b = X;

(b) Trailing fences are necessary. Disallowed outcome $a = b = 0$. Removing a DMBFF after either RMW in Arm would erroneously allow that outcome.

Figure 12: Role of DMBFF fences in IR to Arm mapping. In Arm, the intermediate DMBFF fences restrict the outcomes. Any weaker (or no) fence would allow these outcomes in Arm and the translations would be incorrect.

x86 to IR to Arm. In Figure 10c, we compose the x86→IR (Figure 10a) and IR→Arm mapping (Figure 10b), which is correct:

▸ **Theorem 5** (x86→Arm Correct)**.** The x86→Arm mapping is correct (Definition 3).

Although we had to prove correctness of the mapping components in Agda, correctness of its composition (Theorem 5) follows trivially.

▸ **Theorem 6** (x86→Arm Precise)**.** The x86→Arm mapping is precise (Definition 4).

Precision (Theorem 6) does not follow trivially[13], which we thus demonstrate separately. We show precision by an argument similar to that of the mapping components. Consider the x86 and Arm programs in Figure 11a and Figure 11c. Removing either DMBLD or DMBST fence in Arm would erroneously allow the behavior a=1,b=0 thus both those fences are necessary. To show the DMBFF fences are necessary, we can follow a similar example as shown in Figure 12. Hence the overall x86→Arm mapping is also precise.

[13] Precision does not always follow. Consider a x86→SC→Arm mapping. The x86→SC component must order W−R pairs. The composed mapping unnecessarily orders those too, making it *not* precise.

2.6.2 *Correctness of Optimizing Transformations*

We also study the correctness of various transformations on Lasagne IR programs. We also prove these transformations correct (Definition 3). In contrast to the mapping, for the transformations, both the source and target program execute in the same memory model, which is our LIMM model in this case.

Reorderings. We study the correctness of the reordering adjacent shared memory accesses and fences a and b. In Figure 13, we mark the safe (✓) and unsafe (✗) reorderings. The non-atomic accesses can reorder freely with each other, which LLVM frequently does in its existing optimization passes. Non-atomic accesses *cannot* reorder with an $\mathtt{RMW}_{\mathtt{sc}}$ operation, as that would require reordering with the events it generates upon success and failure. Upon $\mathtt{RMW}_{\mathtt{sc}}$ success, the accesses must reorder with both rmw-related events (*i.e.*, R_{SC} and W_{SC}), which is disallowed by the reordering rules in Figure 13 – in both directions, for both R_{NA} and W_{NA}, the reordering is disallowed.

A store can safely reorder with a succeeding $\mathsf{F}_{\mathtt{rm}}$, a load can safely reorder with a preceding *and* succeeding $\mathsf{F}_{\mathtt{ww}}$. In both cases, the reordering is possible because the fence does not enforce any ordering with the corresponding event. We also observe any pair of fences can reorder safely, as fences do not enforce any order among each other. We prove these reorderings correct (Definition 3) in Agda. In those proofs, we show that a reordering does not remove any 'ord' relation from the target while defining the corresponding source execution.

Memory Access Eliminations. Figure 14 lists the safe access elimination transformations for Read-After-Read (RAR), Read-After-Write (RAW), and Write-After-Write (WAW). A RAR or RAW transformation eliminates a read access by reusing a previously read or written value. In the WAW transformation, the first write is redundant –because it is overwritten by the second– and can safely be eliminated. In these three transformations, the shared memory accesses are po-adjacent. In the subsequent

$\downarrow a \setminus b \rightarrow$	R_{NA}	W_{NA}	R_{SC}	$\mathsf{R}_{\mathsf{SC}} \cdot \mathsf{W}_{\mathsf{SC}}$	F_{RM}	F_{WW}	F_{SC}
R_{NA}	✓	✓	✓	✗	✗	✓	✗
W_{NA}	✓	✓	✓	✗	✓	✗	✗
R_{SC}	✗	✗	✗	✗	✓	✓	✓
$\mathsf{R}_{\mathsf{SC}} \cdot \mathsf{W}_{\mathsf{SC}}$	✗	✗	✗	✗	✓	✓	✓
F_{RM}	✗	✗	✗	✓	=	✓	✓
F_{WW}	✓	✗	✓	✓	✓	=	✓
F_{SC}	✗	✗	✗	✓	✓	✓	=

Figure 13: Reorderings $a \cdot b \rightsquigarrow b \cdot a$ in LIMM. Correct reorderings are marked with ✓. All reorderings are between events on different locations. $a \cdot b$ denotes that a and b are the labels of events related by $\mathsf{po}_{\mathsf{imm}}$. R_{SC} on its own represents a *failed* $\mathtt{RMW}_{\mathtt{sc}}$ read, while '$\mathsf{R}_{\mathsf{SC}} \cdot \mathsf{W}_{\mathsf{SC}}$' corresponds to a *successful* $\mathtt{RMW}_{\mathtt{sc}}$.

three transformations, the memory accesses are separated only by a fence. Those same eliminations are not correct across every kind of fence. We primarily consider these transformation for IR programs obtained by lifting x86 programs with our mapping schemes (Figure 10a).

- F-RAR – We can sometimes eliminate a read after another read from the same location, with a fence in-between, for instance:

  $$`a{=}\mathtt{X};\ \mathtt{F_{rm}};\ b{=}\mathtt{X};' \longrightarrow `a{=}\mathtt{X};\ \mathtt{F_{rm}};\ b{=}a;'$$

  The $\mathtt{F_{rm}}$ fence is the only fence that can occur between two R events, for programs obtained with our x86→IR mapping. We thus prove this transformation across a $\mathtt{F_{rm}}$ fence for a x86-lifted program. In particular, for this proof we specifically require every R event to be followed by a F_{RM} event – which our mappings produce (Figure 10a). To prove this, we construct a source execution where the R events generated by assignments to a and b read from (rf) the same W event; then we show that source execution is well-formed and LIMM-consistent (*i.e.*, it satisfies Coherence, Atomicity, and GOrd).

  A $\mathtt{F_{ww}}$ fence cannot occur in-between R events, as our x86→IR mappings never produce such programs. Surprisingly, this transformation is correct in general, *i.e.*, also for programs not lifted from x86. We do not need to prove this explicitly, as it follows from one of the reordering rules (Figure 13) followed by the regular RAR elimination:

  $$`a{=}\mathtt{X};\ \mathtt{F_{ww}};\ b{=}\mathtt{X};' \xrightarrow{\text{reorder}} `a{=}\mathtt{X};\ b{=}\mathtt{X};\ \mathtt{F_{ww}};' \xrightarrow{\text{RAR}} `a{=}\mathtt{X};\ b{=}a;\ \mathtt{F_{ww}};'$$

  For a $\mathtt{F_{sc}}$ fence this transformation is *incorrect*. We first note that our x86→IR mapping can never produce a program with a $\mathtt{F_{sc}}$ fence in-between two read accesses, making it of limited use for LASAGNE in its current x86-to-Arm context. However, we could attempt to prove this optimization correct in general – for any IR program, including those not lifted from x86. Unlike with the $\mathtt{F_{ww}}$ fence, $\mathtt{F_{sc}}$ cannot safely reorder with non-atomic memory accesses (Figure 13). In particular, we discovered the RAR elimination across a $\mathtt{F_{sc}}$ is not correct in general, of which Figure 15 shows a counterexample.

$\mathsf{R}(X,v) \cdot \mathsf{R}(X,v')$	$\rightsquigarrow$	$\mathsf{R}(X,v)$	(RAR)
$\mathsf{W}(X,v) \cdot \mathsf{R}(X,v)$	$\rightsquigarrow$	$\mathsf{W}(X,v)$	(RAW)
$\mathsf{W}(X,v) \cdot \mathsf{W}(X,v')$	$\rightsquigarrow$	$\mathsf{W}(X,v')$	(WAW)
$\mathsf{R}(X,v) \cdot \mathsf{F}_o \cdot \mathsf{R}(X,v')$	$\rightsquigarrow$	$\mathsf{R}(X,v) \cdot \mathsf{F}_o$	(F-RAR)
$\mathsf{W}(X,v) \cdot \mathsf{F}_\tau \cdot \mathsf{R}(X,v)$	$\rightsquigarrow$	$\mathsf{W}(X,v) \cdot \mathsf{F}_\tau$	(F-RAW)
$\mathsf{W}(X,v) \cdot \mathsf{F}_o \cdot \mathsf{W}(X,v')$	$\rightsquigarrow$	$\mathsf{F}_o \cdot \mathsf{W}(X,v')$	(F-WAW)

Figure 14: Eliminations where $o \in \{\text{RM}, \text{WW}\}$ and $\tau \in \{\text{SC}, \text{WW}\}$. $a \cdot b$ denotes that a and b are the labels of $\mathsf{po_{imm}}$-related events.

- F-RAW – We can sometimes eliminate a read after a write to the same location, with a fence in-between, for instance:

  ‘X=a; F$_{\text{sc}}$; b=X;’ → ‘X=a; F$_{\text{sc}}$; b=a;’

  To prove this in Agda, we assume every R is followed by a F$_{\text{rm}}$ fence and every W event is preceded by a F$_{\text{ww}}$ fence – which our mappings produce (Figure 10a). We construct the source such that the R always reads-from (rf) the W. In the target, we have diverted any GOrd cycle from the eliminated R to the preceding W.

  A F$_{\text{ww}}$ fence cannot occur in-between two W–R events following our mapping, but we can prove this transformation in general with a reordering rule:

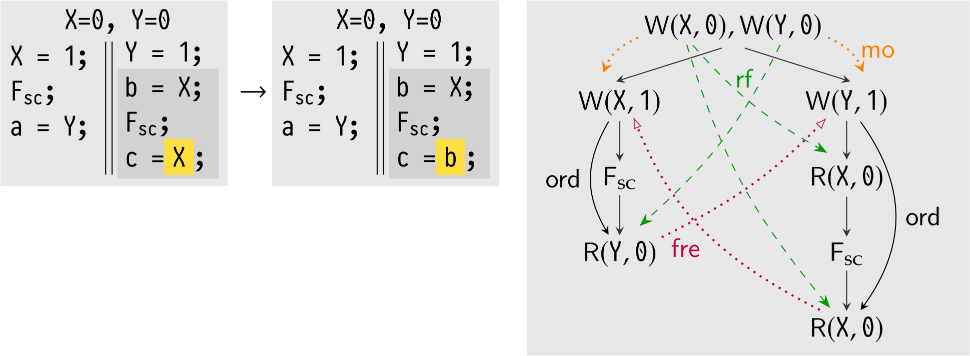
  ‘X=a; F$_{\text{ww}}$; b=X;’ $\xrightarrow{\text{reorder}}$ ‘X=a; b=X; F$_{\text{ww}}$;’ $\xrightarrow{\text{RAW}}$ ‘X=a; b=a; F$_{\text{ww}}$;’

  The read-after-write elimination across a F$_{\text{rm}}$ fence, which cannot appear following our mapping, is also not correct in general.

- F-WAW – Finally, we can eliminate a write after a write to the same location, with a fence in-between, for instance:

  ‘X=a; F$_{\text{ww}}$; X=b;’ → ‘F$_{\text{ww}}$; X=b;’

  To prove this in Agda, we assume every R is followed by a F$_{\text{rm}}$ fence and every W event is preceded by a F$_{\text{ww}}$ fence – which our mappings produce (Figure 10a). Our proofs divert any GOrd cycle from the first W to the second W, which we restrict to be mo-related in the source, meaning the second overwrites the first.



```
      X=0, Y=0                  X=0, Y=0
X = 1;  ||  Y = 1;        X = 1;  ||  Y = 1;
Fsc;    ||  b = X;   →    Fsc;    ||  b = X;
a = Y;  ||  Fsc;          a = Y;  ||  Fsc;
        ||  c = X;                ||  c = b;
```

(a) Example program in LASAGNE IR on which we apply the F-RAR transformation.

(b) The corresponding execution graph. Removing the final R in the right thread breaks the ‘ord;fre;ord;fre’ cycle, erroneously allowing a=0,c=0.

Figure 15: Counterexample for read-after-read elimination across a F$_{\text{sc}}$ fence in LASAGNE IR. By assuming both reads observe the same W event, we implicitly make an incorrect assumption about their memory ordering.

Again, although a $\mathtt{F_{rm}}$ cannot occur in-between two W–W events following our mapping, we can prove it in general with a reordering rule:

$$\text{‘}\mathtt{X}{=}a\mathtt{;}\ \mathtt{F_{rm}}\mathtt{;}\ \mathtt{X}{=}b\mathtt{;}\text{’} \xrightarrow{\text{reorder}} \text{‘}\mathtt{F_{rm}}\mathtt{;}\ \mathtt{X}{=}a\mathtt{;}\ \mathtt{X}{=}b\mathtt{;}\text{’} \xrightarrow{\text{WAW}} \text{‘}\mathtt{F_{rm}}\mathtt{;}\ \mathtt{X}{=}b\mathtt{;}\text{’}$$

Finally, the write-after-write elimination across a $\mathtt{F_{sc}}$ fence, which cannot appear following our mapping, is also not correct in general.

Fence Merging. We can safely merge a fence with an adjacent same or stronger fence. It is also safe to strengthen an $\mathtt{F_{rm}}$ or $\mathtt{F_{ww}}$ fence to a full fence $\mathtt{F_{sc}}$. So given a pair of adjacent $\mathtt{F_{rm}}$ and $\mathtt{F_{ww}}$ fences, we can strengthen and merge them to create one $\mathtt{F_{sc}}$, *i.e.,* $\mathsf{F_{RM}} \cdot \mathsf{F_{WW}} \rightarrow \mathsf{F_{SC}} \cdot \mathsf{F_{SC}} \rightarrow \mathsf{F_{SC}}$. Note, however, that merging fences may break the assumptions that we rely upon in the elimination proofs (*i.e.,* in F-RAR, F-RAW, and F-WAW) as some R events may no longer be followed by a $\mathsf{F_{RM}}$ nor some W events preceded by a $\mathsf{F_{WW}}$ event. In general, the fence elimination optimization should be applied before merging (or reordering). Sometimes we can chain eliminations with merging fences:

$$\begin{array}{lclcl}
\mathsf{R;F_{RM};R;F_{RM}} & \xrightarrow{\text{F-RAR}} & \mathsf{R;F_{RM};F_{RM}} & \xrightarrow{\text{merge}} & \mathsf{R;F_{RM}} \\
\mathsf{F_{WW};W;F_{SC};R;F_{RM}} & \xrightarrow{\text{F-RAW}} & \mathsf{F_{WW};W;F_{SC};F_{RM}} & \xrightarrow{\text{merge}} & \mathsf{F_{WW};W;F_{SC}} \\
\mathsf{F_{WW};W;F_{WW};W} & \xrightarrow{\text{F-WAW}} & \mathsf{F_{WW};F_{WW};W} & \xrightarrow{\text{merge}} & \mathsf{F_{WW};W}
\end{array}$$

2.6.3 *Proof Strategy*

We mechanize the correctness proofs (Definition 3) for the non-trivial transformations in Agda (Agda Team, 2025a), totaling roughly 12,000 lines. We follow the general proof structure from Section 1.4, where our implementation adheres to the following steps: Given a $\mathsf{M^t}$-consistent execution X_t of $\mathbb{P}^\mathsf{t}$, we *(1)* define a source execution X_s from $\mathbb{P}^\mathsf{s}$. Following the mapping scheme, the memory accesses in X_s have corresponding accesses in X_t. Then, we *(2)* relate the X_s and X_t relations that are used in $\mathsf{M^s}$ and $\mathsf{M^t}$ and show that the X_s is well-formed. Similarly, we *(3)* then show that X_s satisfies the axioms in $\mathsf{M^s}$. Finally, we *(4)* show the terminal value of $\mathsf{X}_t.\mathsf{mo}$ and $\mathsf{X}_s.\mathsf{mo}$ match, for each memory location, meaning X_t and X_s have identical behaviors. In mapping schemes the source and target models differ and for the transformations on the IR both $\mathsf{M^s}$ and $\mathsf{M^t}$ are LIMM model.

2.7 IMPLEMENTATION

Lasagne is implemented on top of Microsoft's mctoll binary lifting tool (Yadavalli and Smith, 2019) and the LLVM compiler framework (LLVM Team, 2025; Lattner and Adve, 2004), both open-source projects. We enforce x86 to IR mapping from Figure 10a on the lifted code, for which we insert fences in two steps:

1. For every load and store, we explore the use-def chain of their pointer operand. In this exploration, we ignore bitcast and `getelementptr` operations, looking for a potential stack allocation. If the operations are thread-local, *e.g.*, by addressing the stack, then no fence is inserted. Otherwise, the access is conservatively treated as a shared memory access and fences are inserted following the mapping scheme from Figure 10a.
2. We merge pairs of fences in the same basic block if there are no memory-access instructions in between.

After placing fences, we apply the existing LLVM optimizations. These optimizations are essential to eliminate unnecessary code produced while lifting. We implement the IR→Arm scheme (Figure 10b) in the LLVM backend that generates Arm code.

2.8 EVALUATION

We evaluate Lasagne with the Phoenix benchmark suite (Ranger et al., 2007) on a 16-core Arm Cortex-A72 with 32 GiB of RAM. We expect the x86→Arm translated programs to perform worse than natively-compiled Arm programs, as the translation incurs overhead. After lifting the program, correctly inserting the fences (Figure 10a), and compiling it to Arm (Figure 10b) the programs take 1.67× of the native execution time, on average. After applying lifting-specific refinements and optimizations –enabled by our translation rules from Section 2.6.2– we observe only 1.51× the native execution time, on average. Our paper (Rocha et al., 2022) further details Lasagne's implementation and experiments[14].

2.9 CONCLUSION

In this chapter, we presented Lasagne, a static binary translator for weak memory model architectures. Lasagne can lift x86 binaries to LLVM IR and then compile it to Arm while enforcing the x86-TSO weak memory model. We provided formally verified mappings from x86 to LLVM IR—which follows our LIMM model—to Arm and transformations on the IR, which include transformations occurring in existing LLVM optimizations. Lasagne's mapping schemes enforce x86-Arm robustness in general without inserting unnecessary fences.

Future Work. As a static translator, Lasagne cannot translate all programs. Some information was lost when compiling the original x86 program (Andriesse et al., 2016), which we cannot always recover statically (Rice, 1953). Dynamic binary translation is an alternative approach, which avoids this limitation by translating programs dynamically. In Chapter 3, we present our dynamic binary translator Risotto.

Lasagne's x86→Arm mappings translate x86's `RMW` instruction to an Arm ll/sc instruction pair. As those order weakly, our mappings had to insert expensive `DMBFF`

[14] As my collaborators worked on Lasagne's system and experiments, I omitted them from this dissertation

fences *before and after* them. Recently, Alglave et al. (2021) proposed a new Arm model with single-instruction `CAS` operations. In particular, the $\texttt{CAS}_{\texttt{AL}}$ variant aims to order like x86's `RMW`, meaning it does not need the surrounding `DMBFF` fences, benefiting performance. We also study those new instructions in Chapter 3.

The same paper by Alglave et al. (2021) proposed a mixed-size memory model (Flur et al., 2017), where memory is accessed in units of multiple sizes (*e.g.,* 1/2/4/8-bytes). Lasagne's mappings do not consider those models, meaning we cannot guarantee correct translation of mixed-size accesses. We study those in Chapter 4.

Finally, Lasagne's proofs were mechanized in Agda, resulting in a code base of roughly 12,000 lines. Those mechanized proofs are very *rigid*, where making changes—*e.g.,* to add features—is very time-consuming. As much structure is shared between proofs, they would benefit from a general framework capturing that structure. Then programmers would not spend countless hours on domain-irrelevant mechanization details, but only inspect the proof cases relevant to their domain. In Chapter 5 we address that challenge.

Chapter 3

Dynamic Program Translation with Mapping Schemes

ABSTRACT Dynamic Binary Translation (DBT) is a powerful approach to support cross-architecture emulation of unmodified binaries. However, DBT systems face correctness challenges when emulating *concurrent binaries from strong to weak memory consistency architectures*. We show that QEMU, a state-of-the-art emulator, contains several translation errors, when emulating x86 binaries on Arm hosts.

To address that challenge, we propose an end-to-end approach that correctly and efficiently emulates weak memory model architectures. Our contributions are twofold: First, we formalize QEMU's intermediate representation (TCG IR) memory model, and use it to propose formally verified mapping schemes bridging the *strong-on-weak memory consistency mismatch*. Second, we implement these verified mappings in Risotto, a QEMU-based DBT system that optimizes memory fence placement while ensuring correctness.

3.1 INTRODUCTION

Dynamic Binary Translation (DBT) systems emulate the program's *guest* ISA on the *host* machine, by translating the code at run time (QEMU Team, 2003; Bellard, 2005; Spink et al., 2019). A major challenge for DBT systems is correct and performant emulation of concurrent binaries (Cota et al., 2017; Lustig et al., 2015).

Performance. In contrast to Static Binary Translation (SBT), which necessarily fails to translate some programs, as we saw with Lasagne (Chapter 2), DBT can translate any program because it does not recover semantic information statically (*e.g.*, dynamic jump targets). A dynamic translator naturally encounters that information for any specific program context at runtime. Although dynamic translation techniques can translate any program, they often incur a significant performance overhead in comparison to static translation. Hence, a primary challenge is to reduce that performance overhead.

Correctness. Another challenge is ensuring that the translation between architectures is correct, which is particularly problematic when translating from a strong memory model, *e.g.*, x86 (Owens et al., 2009), to a weaker model, *e.g.*, Arm (Alglave et al., 2021). Like our static translator Lasagne (Chapter 2), a dynamic translator must ensure that the behavior of the guest ISA is correctly reproduced on the host machine, for instance, by placing memory fences. The state-of-art DBT system QEMU (QEMU Team, 2003; Bellard, 2005) does not officially support strong-on-weak emulation (QEMU Team, 2021) but still inserts fences to preserve the stronger memory orders. In particular, it attempts to enforce a stronger order than x86's when emulating on Arm, which *would be* correct but may harm performance (Liu et al., 2020). However, despite its attempt at enforcing a strong ordering, it fails to do that correctly—we discover and report several translation errors in QEMU caused by incorrectly placed fences that may lead to errors at run time. Further, while proving mapping correctness, we discover and report that the Arm memory model (Alglave et al., 2021) orders its new $\texttt{CAS}_{\texttt{AL}}$ instruction too weakly.

Solution. To address the above issues, in this chapter, we propose an end-to-end DBT approach based on QEMU that executes concurrent x86 binaries correctly and efficiently on Arm architectures by combining: formal verification of translation correctness for strong-on-weak architecture and a DBT system for runtime binary translation based on those verified translation rules.

More specifically, we propose the *first formal concurrency memory model* of QEMU's intermediate representation (TCG IR). We use that model to define and prove correct the mapping schemes from *(1)* x86 to TCG IR and *(2)* TCG IR to Arm. We mechanize the proofs for these mapping schemes in the Agda proof assistant (Agda Team, 2025a). In contrast to LASAGNE's mappings (Chapter 2), which used Arm's older ll/sc RMW instruction pair, these new mappings target Arm's new single-instruction $\texttt{CAS}_{\texttt{AL}}$ ("Compare and Swap Acquire Release") instruction.

Another aspect of QEMU's Tiny Code Generator (TCG) is its intermediate optimizations on concurrency primitives, which may affect translation correctness, as transformation for sequential programs may be not be correct for concurrent programs (Morisset et al., 2013; Vafeiadis et al., 2015; Chakraborty and Vafeiadis, 2016). Only ensuring the memory model mismatches in architectures (Cota et al., 2017; Lustig et al., 2015) does not guarantee correct translation in QEMU. Therefore, we prove the correctness of various transformations that are commonly used by TCG's optimizations—these are similar to those used by the LASAGNE's LLVM optimizations (Chapter 2). These verified transformations, along with verified mappings, facilitate the development of our end-to-end DBT system RISOTTO, based on QEMU. Overall, this chapter makes the following contributions:

- **Concurrency analysis in QEMU and the Arm memory model.** We discover and report several translation errors in QEMU due to incorrectly placed memory fences. We also report *undesired behavior* in the Arm memory model (Alglave et al., 2021) –herein referred to as Armed-Cats– for efficient x86-to-Arm translation, and propose revisions to the model for verified mappings. An equivalent revision was accepted for that model (Alglave, 2022).
- **TCG IR memory model: Formalization, verified mappings and optimizations.** We formalize the memory model of QEMU's TCG IR. Based on this formal model, we propose mapping schemes from x86 to TCG IR and TCG IR to Arm, which we

verify to be semantically correct. We implement those mapping schemes in our Risotto DBT system. We also prove the correctness of various optimizations on TCG IR model that are performed by QEMU.

3.2 BACKGROUND

3.2.1 *Dynamic Binary Translation*

DBT systems typically operate as follows: they *(1)* translate the instruction currently pointed at by the emulated Instruction Pointer (IP), and *(2)* execute the translated instruction, updating the IP to either the following instruction or the target of a jump. Most DBTs implement translation granularities of at least a *basic block*, and employ classic compiler optimizations to improve generated code quality (and hence run time performance). Basic blocks are often cached to avoid repeating translations.

3.2.2 *TCG: QEMU's Dynamic Binary Translator*

QEMU is a state-of-the-art emulator capable of cross-ISA emulation, which translates code through its TCG. Basic blocks are translated via an IR called the TCG IR. Architecture-independent optimizations are applied on basic blocks at the IR level.

TCG IR. The TCG IR is an assembly-like instruction set, containing basic arithmetic, logic, and control flow instructions. However, floating-point arithmetic is emulated via integer-based computations.

Memory fences. The TCG IR provides fences for all types of pairs of accesses. For example, the $\mathtt{F}_{\mathtt{ww}}$ fence orders a store-store pair, while $\mathtt{F}_{\mathtt{rw}}$ orders a load-store pair. When generating fences in the IR, TCG takes the guest memory model into account to choose the fence accordingly. Section 3.3 provides a more detailed discussion.

Atomic read-modify-write (RMW). RMW accesses are currently translated into calls to helper functions in QEMU. Therefore, even if the host ISA has an equivalent atomic instruction, execution is still transferred from the emulated binary to QEMU. We discuss these primitives in Section 3.5.

Optimizations. TCG performs various optimizations on the translated basic blocks at the IR level. Some of the well-known optimizations are dead code elimination, constant propagation and folding, and consecutive fence merging.

3.2.3 *Concurrency Primitives in Architectures*

In Section 1.3 we have already explained the concurrency primitives and semantics of x86 and Arm. Here, we briefly elaborate on the subtleties of RMW instructions. In particular, Arm provides two kinds, which we denote by CAS and RMW[2].
CAS represents the newer single-instruction read-modify-write instruction (Arm, 2016; Alglave et al., 2021), which has multiple variants that order differently. For instance,

Access type	x86	TCG IR	Arm
Load	`RMOV`	`ld`	`LDR`
Store	`WMOV`	`st`	`STR`
Full-fence	`MFENCE`	$\mathtt{F}_{\mathtt{sc}}$	`DMBFF`
WW-fence		$\mathtt{F}_{\mathtt{ww}}$	`DMBST`
RM-fence		$\mathtt{F}_{\mathtt{rm}}$	`DMBLD`
MR-fence		$\mathtt{F}_{\mathtt{mr}}$	
MW-fence		$\mathtt{F}_{\mathtt{mw}}$	
Atomic-update	`CMPXCHG`	`RMW`	`CAS`, $\mathtt{RMW}^2$
Rel.Acq. atomic-update			$\mathtt{CAS}_{\mathtt{AL}}$, $\mathtt{RMW}^2_{\mathtt{AL}}$

$$\mathtt{RMW}^2 \triangleq \ell :\ \mathsf{LX};\ \mathsf{cmp};\ \mathsf{bc}\ \ell';\ \mathsf{SX};\ \mathsf{bc}\ \ell;\ \ell' :$$

$$\mathtt{RMW}^2_{\mathtt{AL}} \triangleq \ell :\ \mathsf{LX}_{\mathsf{A}};\ \mathsf{cmp};\ \mathsf{bc}\ \ell';\ \mathsf{SX}_{\mathsf{L}};\ \mathsf{bc}\ \ell;\ \ell' :$$

Figure 16: Concurrency primitives in x86, TCG IR, and Arm that appear in the mapping schemes. For fences, 'R' = read, 'W' = write, 'M' = memory – *e.g.*, an RM fence orders *preceding reads* with *succeeding reads and writes*.

$\mathtt{CAS}_{\mathtt{AL}}$ orders with predecessor and successors. $\mathtt{RMW}^2$ consists of *two* instructions: load-exclusive (LX) and store-exclusive (SX) instructions. Arm provides variants of it that order differently, for instance, $\mathtt{RMW}^2_{\mathtt{AL}}$ consists of acquire-load-exclusive (LX_{A}) and release-store-exclusive (SX_{L}) instructions. A release access is ordered with its predecessors and an acquire is ordered with its successors. We can thus construct several variants with those instructions: $\mathtt{RMW}^2$, $\mathtt{RMW}^2_{\mathtt{A}}$, $\mathtt{RMW}^2_{\mathtt{L}}$, $\mathtt{RMW}^2_{\mathtt{AL}}$. Similar to $\mathtt{RMW}^2$, $\mathtt{CAS}_{\mathtt{AL}}$ accesses can also have release/acquire combinations. In x86, a successful `CMPXCHG` acts as a full fence, whereas in Arm only a successful $\mathtt{CAS}_{\mathtt{AL}}$ acts as a full fence. Figure 16 includes an overview of concurrency primitives in x86, TCG IR, and Arm.

3.3 MOTIVATION

We expose correctness and performance problems that arise when QEMU emulates concurrent programs. We also expose an error in an existing Arm mapping.

3.3.1 *Emulation of Concurrent Programs in QEMU*

QEMU does not officially support the emulation of strongly ordered ISAs, *e.g.*, x86, on weakly ordered ones, *e.g.*, Arm. However, in user mode emulation, the program runs without triggering any warning or error message to users, one may therefore think that support is available.

QEMU mapping schemes. Figure 17 shows QEMU's mapping schemes for translating memory-related x86 instructions to Arm. An $\mathtt{F}_{\mathtt{mr}}$ fence is inserted before loads

x86		TCG IR		Arm
RMOV	→	$\texttt{F}_{\texttt{mr}}$; ld	→	DMBLD; LDR
WMOV	→	$\texttt{F}_{\texttt{mw}}$; st	→	DMBFF; STR
RMW	→	call	→	BLR; RMW; RET
MFENCE	→	$\texttt{F}_{\texttt{sc}}$	→	DMBFF

Figure 17: QEMU's existing mappings from x86 to Arm, which are incorrect.

(ld), ordering the load with its preceding memory access. Since store-load reordering is allowed in x86, TCG demotes this fence to $\texttt{F}_{\texttt{rr}}$, only ordering the load with a preceding load. This is an attempt to match the x86 memory model. An $\texttt{F}_{\texttt{mw}}$ fence is inserted before stores (st), ordering the store with its preceding memory access. These fences are then lowered to Arm's DMBLD and DMBFF fences.

RMW **operations**. QEMU translates RMW operations as calls to helper functions. These helper functions rely on GCC built-ins for the atomic accesses. As a result, depending on the GCC version, the instructions differ. For example, the helper function emulating the x86 CMPXCHG instruction uses an $\texttt{LX}_{\texttt{A}}$-$\texttt{SX}_{\texttt{L}}$ pair ($\texttt{RMW}^{2}_{\texttt{AL}}$) with GCC 9, but a $\texttt{CAS}_{\texttt{AL}}$ instruction ($\texttt{RMW}^{1}_{\texttt{AL}}$) with GCC 10. Both are correct from GCC's standpoint since they both comply with the C/C++ memory model, but order differently on x86 (*i.e.*, $\texttt{CAS}_{\texttt{AL}}$ is stronger).

3.3.2 *Correctness: Errors in QEMU*

We found several errors in QEMU's x86 to Arm translation, more specifically in handling RMW access (both CAS and $\texttt{RMW}^{2}$). We demonstrate these errors with the translations of the MPQ and SBQ programs where CAS and $\texttt{RMW}^{2}$ accesses are generated, respectively. We also show that using a $\texttt{F}_{\texttt{mr}}$ fence in TCG IR may also result in an erroneous read-after-write transformation as demonstrated by the translation of the FMR program.

Error in mapping scheme with $\texttt{CAS}_{\texttt{AL}}$. Consider the x86 to Arm mapping by QEMU for the following program.

```
       X = Y = 0;
X = 1; || a = Y;
Y = 1; || if (a == 1)
       ||     RMW(X, 1, 2);
```

⇝

```
        X = Y = 0;
DMBFF; || DMBLD;
X = 1; || a = Y;
DMBFF; || if (a == 1)
Y = 1; ||     CAS_AL(X, 1, 2);
```

(MPQ)

In x86, a=1 implies that all writes in the first thread are completed. Since reads are not reordered, the RMW always reads the X=1 and successfully updates X=2. As a result a=1, X=1 is never possible. In Arm, however, a read and a read-acquire pair can be reordered. This means that even though the first thread's writes are ordered by fences, the read of $\mathtt{CAS}_{\mathtt{AL}}$ can be speculatively executed before the a=Y instruction as they are unordered. In that case, the $\mathtt{CAS}_{\mathtt{AL}}$ will not observe X=1 and fail, but the result will still be committed after a=Y sets a to 1. It results in the outcome a=1, X=1, which is disallowed in x86, thus demonstrating that the translation is incorrect.

Error in mapping scheme with $\mathtt{RMW}^2_{\mathtt{AL}}$. Consider the following x86 to Arm translation.

```
  X = Y = Z = U = 0;                  X = Y = Z = U = 0;
X = 1;       || Y = 1;          DMBFF;                || DMBFF;
RMW(Z,0,1);  || RMW(U,0,1);  ⇝  X = 1;                || Y = 1;                  (SBQ)
a = Y;       || b = X;          RMW$^2_{AL}$(Z,0,1);  || RMW$^2_{AL}$(U,0,1);
                                DMBLD;                || DMBLD;
                                a = Y;                || b = X;
```

The behavior in question is Z=U=1, a=b=0. In x86, successful CMPXCHG accesses order store-load access pairs across it. On the other hand, neither successful $\mathtt{RMW}^2_{\mathtt{AL}}$ accesses nor DMBLD fences can order the store-load access pairs. Thus, the mapping results in a new outcome in the generated Arm program and, therefore, the overall translation is incorrect.

Error in RAW transformation in TCG IR. QEMU performs various constant propagation optimizations on TCG IR such as read-after write (RAW), *e.g.*, 'Y=2; a=Y; ⇝ Y=2; a=2;'. In the presence of $\mathtt{F}_{\mathtt{mr}}$, the RAW transformation is incorrect as it introduces a new outcome. Consider the following example:

```
    X = Y = Z = 0;                          X = Y = Z = 0;
X = 3;    || if (Z == 2) {              X = 3;    || if (Z == 2) {
F_mr;     ||     F_rw;          RAW     F_mr;     ||     F_rw;
Y = 2;    ||     X = 4;        ----->   Y = 2;    ||     X = 4;          (FMR)
a = Y;    ||     c = X;                 a = 2;    ||     c = X;
F_rw;     || }                          F_rw;     || }
Z = 2;    ||                            Z = 2;    ||
```

Consider the outcome a=2, c=3. In the source TCG program, the $\mathtt{F}_{\mathtt{mr}}$ and $\mathtt{F}_{\mathtt{rw}}$ fences in the first thread establish dependency-based ordering from X=3 to Z=2 via a=Y. In the second thread, $\mathtt{F}_{\mathtt{rw}}$ orders the read of Z with the successor accesses on X. As a result, the outcome a=2,c=3 is disallowed. The RAW transformation in the first thread removes the read of Y and hence X=3 and Z=2 are not ordered anymore. As a result, the a=2, c=3 outcome is allowed in target program, making the RAW transformation incorrect.

x86		Arm
`RMOV`	→	`LDAPR`
`WMOV`	→	`STLR`
`CMPXCHG`	→	$\texttt{CAS}_{\texttt{AL}}$
`MFENCE`	→	`DMBFF`

Figure 18: Arm mapping of Armed-Cats (Alglave et al., 2021, §2.5), which is incorrect.

3.3.3 *Correctness: Error in "Desired" Arm Mapping*

We consider the x86 to Armed-Cats mapping by Alglave et al. (2021, §2.5). While they do *not explicitly* give a mapping, we infer:

- `LDAPR` ($\texttt{LDR}_{\texttt{Q}}$) and `STLR` ($\texttt{STR}_{\texttt{L}}$) enable efficient emulation of x86-TSO on Armed-Cats (Alglave et al., 2021, p.6)
- amo in Armed-Cats exclusively models `CAS` operations, *e.g.*, $\texttt{CAS}_{\texttt{AL}}$, which *should act "as a full barrier"* (Alglave et al., 2021, p.18).
- In x86, a successful RMW (*i.e.*, `CMPXCHG`) also behaves like a full barrier (Owens, 2010; Alglave et al., 2021).

We interpret that mapping as shown in Figure 18. While examining it, we discover the mapping is incorrect following the memory models Alglave et al. (2021). Consider the following example:

$$\begin{array}{c}\texttt{X = Y = 0;}\\ \begin{array}{l||l} \texttt{CMPXCHG(X,0,1);} & \texttt{CMPXCHG(Y,0,1);}\\ a\ \texttt{= Y;} & b\ \texttt{= X;}\end{array}\end{array} \xrightarrow{\text{Armed-Cats}} \begin{array}{c}\texttt{X = Y = 0;}\\ \begin{array}{l||l} \texttt{CAS}_{\texttt{AL}}\texttt{(X,0,1);} & \texttt{CAS}_{\texttt{AL}}\texttt{(Y,0,1);}\\ a\ \texttt{=}\ \texttt{Y}_{\texttt{Q}}\texttt{;} & b\ \texttt{=}\ \texttt{X}_{\texttt{Q}}\texttt{;}\end{array}\end{array} \quad \text{(SBAL)}$$

The notation '$\texttt{X}_{\texttt{Q}}$' means we read from `X` with the `LDAPR` ($\texttt{LDR}_{\texttt{Q}}$) Arm instruction. The source x86 program *disallows* $\texttt{X=Y=1,}a\texttt{=}b\texttt{=0}$ as outcome, while the Arm program *allows* it. Therefore the mapping is wrong.

Fixing this error. There are two options to fix this error in the model:

- Keep the current model and accept $\texttt{CAS}_{\texttt{AL}}$ is insufficient to model x86 `CMPXCHG`; or
- Strengthen the formal model slightly, so $\texttt{CAS}_{\texttt{AL}}$ behaves like x86 `CMPXCHG`.

We choose the second option that we detail in Section 3.5.2. We hypothesize that hardware may already be consistent with our model. We contacted the authors of Armed-Cats, but they could not confirm hardware behavior with regards to our SBAL example in the new model. However, they still decided to strengthen the memory model (Alglave, 2022).

3.3.4 *Performance: Fence placement*

QEMU's mapping schemes in Figure 17 prevent any reordering of memory accesses, even though the guest ISA (x86) allows some reorderings to happen. However, the CPU performs these reorderings to maximize its utilization. Not taking advantage of the CPU's instruction scheduling hurts performance. Additionally, having fences *before* every access makes it impossible to merge them.

3.4 OVERVIEW

We propose an end-to-end approach to improve the performance of strong-on-weak architecture DBT while maintaining semantic correctness.

3.4.1 *Verified Mapping Schemes and Optimizations*

We reason about the end-to-end translation steps: (1) x86 to TCG IR mapping (2) TCG IR to TCG IR optimization (3) TCG IR to Arm mapping.

TCG IR formalization. To reason about these steps formally, we use existing formal models of x86 and Arm (Alglave et al., 2021), and propose a formalization of TCG IR. Based on this formalization, we ensure the translations in all three steps are correct.

Mappings in steps (1) & (3). We map the load, store, RMW, and fence accesses from the source to the corresponding accesses in the target models. The orderings between the accesses vary based on the consistency models. To ensure orderings between weaker accesses, we introduce additional leading or trailing fences along with the memory accesses. As fences are costly, our goal is to introduce only the minimal fences that are required to ensure correctness.

Moreover, we note that some TCG optimizations perform read-after-write (RAW) transformations, which can introduce errors around F_{mr} or F_{wr} fences (see FMR). Hence, we avoid generating any F_{mr} or F_{wr} fence in the x86 to TCG IR mapping scheme so that RAW transformations remain correct on the generated TCG IR programs. Using all three formal models, we prove the mapping schemes correct.

IR transformations in step (2). Risotto performs several optimizations on the TCG IR before generating the Arm code. To ensure correctness, we analyze common transformations performed on concurrency primitives. We show that the proposed TCG IR formalization allows the transformations performed by Risotto's optimizations.

More specifically, we reason about elimination of redundant shared memory accesses and reordering of shared memory accesses. We also reason about fence merging optimizations which can be performed when there are adjacent fences. Our x86 to TCG IR mapping scheme creates such adjacent fences which can be merged to improve performance.

3.4.2 *Risotto: A Dynamic Binary Translator for Strong-on-Weak Architectures*

We build Risotto upon the widely used emulator QEMU. We improve over the existing work through three contributions: *(i)* the implementation of formally verified memory mappings and *(ii)* a fast and correct translation of Compare-and-Swap (CAS) instructions.

Memory mappings. We first replace the memory mapping schemes used by QEMU with our schemes presented in Section 3.5, which are formally verified to enforce the x86 memory model (Owens et al., 2009; Owens, 2010; Alglave et al., 2021). We implement these mapping schemes in QEMU. We also implement fence merging optimizations at the TCG IR level to minimize the cost of inserted fences.

Fast and correct CAS. As previously stated, `RMW` primitives are emulated through a call to a helper function in QEMU and not translated. In addition to the performance hit, this can also trigger erroneous behaviors.

In Risotto, we aim at preserving correctness while maximizing performance. For atomic operations, we propose to translate the x86 atomic instructions, *e.g.,* `CMPXCHG`, directly into Arm assembly, *e.g.,* using the new $\texttt{CAS}_{\texttt{AL}}$ instructions. This allows us to fix the errors in QEMU's current scheme as well as improve performance. We also implement this in the TCG (Section 3.7.2).

3.5 TIMM WEAK MEMORY MODEL

In this section, we propose an axiomatic concurrency model for the TCG IR (TIMM). Based on this model, we propose formally verified mapping schemes from x86 to Arm via the TCG IR. Our axiomatic concurrency models build upon the semantics introduced in Chapter 1.

3.5.1 *Arm Model*

We use the Arm model from Section 1.3.3, but modify[15] it subtly, as shown in Figure 19. We propose our fix to the Armed-Cats model (Alglave et al., 2021), where we explicitly po-order the domain and co-domain of the $\texttt{CAS}_{\texttt{AL}}$-generated amo relation between R_A and W_L events. That resolves the $\texttt{CAS}_{\texttt{AL}}$ problem described in Section 3.3.3 – we now ensure the $\texttt{CAS}_{\texttt{AL}}$ instruction *acts as a full barrier*, like `CMPXCHG` does in x86.

3.5.2 *Formalizing TCG IR Concurrency*

We begin with the TCG primitives with its generated events and relations.

Load and store accesses. TCG provides load (`ld`) and store (`st`) operations that read and write shared memory locations, respectively, generating R and W events.

[15] The model presented in Section 1.3.3 does not include the '$\mathsf{po};[\mathsf{dom}([\mathsf{R}_\mathsf{A}];\mathsf{amo};[\mathsf{W}_\mathsf{L}])]$' case of bob. When we initially proposed our fix, we were not sure whether it was necessary. However, we later discovered that case was not necessary for our mappings (in Chapter 4) and for the Armed-Cats mappings (in Chapter 5). In this chapter, we keep it in the model as it remains in our corresponding Agda proofs.

$$\begin{aligned}
&\mathsf{ob}\text{ is irreflexive} && \text{(External)}\\
&\quad\text{where } \mathsf{ob} \triangleq (\mathsf{obs} \cup \mathsf{lob})^{+}\\
&\qquad\quad \mathsf{obs} \triangleq \mathsf{rfe} \cup \mathsf{moe} \cup \mathsf{fre}\\
&\qquad\quad \mathsf{bob} \triangleq \mathsf{po};[\mathsf{F}];\mathsf{po} \ \cup\ [\mathsf{R}];\mathsf{po};[\mathsf{F}_{\text{LD}}];\mathsf{po} \ \cup\ [\mathsf{W}];\mathsf{po};[\mathsf{F}_{\text{ST}}];\mathsf{po};[\mathsf{W}]\\
&\qquad\qquad\quad \cup\ \mathsf{po};[\mathsf{dom}([\mathsf{R}_{\text{A}}];\mathsf{amo};[\mathsf{W}_{\text{L}}])] \ \cup\ [\mathsf{codom}([\mathsf{R}_{\text{A}}];\mathsf{amo};[\mathsf{W}_{\text{L}}])];\mathsf{po}\\
&\qquad\qquad\quad \cup\ \ldots
\end{aligned}$$

Figure 19: Armed-Cats Arm model (Alglave et al., 2021), corrected (marked green).

Fence accesses. TCG provides different types of fences: $\mathtt{F_{rr}}$, $\mathtt{F_{rw}}$, $\mathtt{F_{ww}}$, $\mathtt{F_{wr}}$, $\mathtt{F_{acq}}$, $\mathtt{F_{rel}}$, and $\mathtt{F_{sc}}$. These fences generate F_{RR}, F_{RW}, F_{WW}, F_{WR}, F_{ACQ}, F_{REL}, and F_{SC} events respectively. They can be combined to define stronger fences, *e.g.*, we combine $\mathtt{F_{rr}}$ and $\mathtt{F_{rw}}$ to define $\mathtt{F_{rm}}$, that generates an F_{RM} event for the proposed mapping schemes. All these fences order certain memory accesses which we capture in *order* (ord) relations. For instance, a pair of po-related events (a, b) are in ord relation if a and b are W events with an intermediate F_{WW} event following the '[W];po;[F_{WW}];po;[W]' rule. While TCG contains both $\mathtt{F_{sc}}$ and $\mathtt{F_{rr}}$ fences, they order identically[16]. Finally, the F_{ACQ} and F_{REL} fences do not enforce any order but remain for legacy reasons.

`RMW` **accesses.** TCG also provides a number of atomic read-modify-write (`RMW`) operations. These atomic `RMW` accesses follow Sequential Consistency (SC) semantics and do not allow reordering with other accesses. A successful `RMW` generates a rmw-related R_{SC} and W_{SC} event pair, *i.e.*, $[\mathsf{R}_{\text{SC}}];\mathsf{rmw};[\mathsf{W}_{\text{SC}}]$. A failed `RMW` generates only a single R_{SC} event. Finally, $\mathsf{R}_{\text{SC}} \subseteq \mathsf{R}$ and $\mathsf{W}_{\text{SC}} \subseteq \mathsf{W}$ hold in the model. Events generated from `RMW` accesses also enforce ord relation as shown in the ord definition.

Finally, we define global-happen-before (ghb) relation to order events across different threads. On an execution graph, $\mathsf{ghb}(a, b)$ implies that there is a path from a to b by ord and external relations rfe, moe, and fre.

Axioms. Based on these relations, we define the consistency constraints. Similar to x86 and Arm, the TCG IR model also includes the Coherence and Atomicity axioms. The GOrd axiom in Figure 20 ensures a global order between events.

3.6 MAPPING SCHEMES

Based on the proposed IR model, we verify the correctness of the transformations (mappings and transformations). In particular, we again prove them correct (Definition 3 from Section 1.4) in the Agda proof assistant (Agda Team, 2025a).

[16] Their ordering rules (in ord) differ syntactically, but they order identically.

$$
\begin{array}{llll}
\multicolumn{3}{l}{\mathsf{ghb} \text{ is irreflexive}} & (\mathsf{GOrd}) \\
\multicolumn{4}{l}{\quad \text{where } \mathsf{ghb} \triangleq (\mathsf{ord} \cup \mathsf{rfe} \cup \mathsf{moe} \cup \mathsf{fre})^{+}} \\
\qquad \mathsf{ord} \triangleq & & [\mathsf{R}];\mathsf{po};[\mathsf{F}_{\mathrm{RR}}];\mathsf{po};[\mathsf{R}] & \cup\ [\mathsf{R}];\mathsf{po};[\mathsf{F}_{\mathrm{RW}}];\mathsf{po};[\mathsf{W}] \\
 & \cup & [\mathsf{R}];\mathsf{po};[\mathsf{F}_{\mathrm{RM}}];\mathsf{po};[\mathsf{R} \cup \mathsf{W}] & \cup\ [\mathsf{W}];\mathsf{po};[\mathsf{F}_{\mathrm{WR}}];\mathsf{po};[\mathsf{R}] \\
 & \cup & [\mathsf{W}];\mathsf{po};[\mathsf{F}_{\mathrm{WW}}];\mathsf{po};[\mathsf{W}] & \cup\ [\mathsf{W}];\mathsf{po};[\mathsf{F}_{\mathrm{WM}}];\mathsf{po};[\mathsf{R} \cup \mathsf{W}] \\
 & \cup & [\mathsf{R} \cup \mathsf{W}];\mathsf{po};[\mathsf{F}_{\mathrm{MR}}];\mathsf{po};[\mathsf{R}] & \cup\ [\mathsf{R} \cup \mathsf{W}];\mathsf{po};[\mathsf{F}_{\mathrm{MW}}];\mathsf{po};[\mathsf{W}] \\
 & \cup & [\mathsf{R} \cup \mathsf{W}];\mathsf{po};[\mathsf{F}_{\mathrm{MM}}];\mathsf{po};[\mathsf{R} \cup \mathsf{W}] & \\
 & \cup & \mathsf{po};[\mathsf{W}_{\mathrm{SC}} \cup \mathsf{dom}(\mathsf{rmw})] & \cup\ [\mathsf{R}_{\mathrm{SC}} \cup \mathsf{codom}(\mathsf{rmw})];\mathsf{po} \\
 & \cup & \mathsf{po};[\mathsf{F}_{\mathrm{SC}}] & \cup\ [\mathsf{F}_{\mathrm{SC}}];\mathsf{po}
\end{array}
$$

Figure 20: Proposed TCG IR memory model (TIMM). TCG also satisfies the Coherence and Atomicity axioms, like x86 and Arm.

Correct mapping schemes. We translate concurrency primitives from x86 to Arm in two steps: *(1)* x86 to TCG IR and *(2)* TCG IR to Arm. We prove these mapping schemes correct (Definition 3) in Agda, and also show they are *precise* (Definition 4).

x86 to IR mapping scheme. We give the mapping scheme in Figure 21a, which introduces fences with the load and store accesses to enforce the same restrictions as x86.

▶ **Theorem 7** (x86→IR Correct). The x86→IR mapping is correct (Definition 3).

We have proved Theorem 7 in Agda.

▶ **Theorem 8** (x86→IR Precise). The x86→IR mapping is precise (Definition 4).

We prove Theorem 8 by showing each fence is necessary in some program. In x86 load-load and load-store accesses are ordered (formally by xppo) unlike that of IR. To enforce these orderings (formally ord) in the generated programs we require the trailing and leading fences with load and store respectively as shown in Figure 22. Notably, although weaker fences are available in the IR, only a $\mathtt{F}_{\mathtt{rr}}$ or $\mathtt{F}_{\mathtt{rw}}$ is insufficient to order the accesses, meaning the $\mathtt{F}_{\mathtt{rm}}$ is necessary after a load.

IR to Arm mapping scheme. We give the mapping schemes in Figure 21b.

▶ **Theorem 9** (IR→Arm Correct). The IR→Arm mapping is correct (Definition 3).

We have proved Theorem 9 in Agda.

▶ **Theorem 10** (IR→Arm Precise). The IR→Arm mapping is precise (Definition 4).

To prove Theorem 10, we analyze the fences in this mapping. If a TCG `RMW` generates $\mathtt{RMW}^2$ access then it introduces leading and trailing `DMBFF` fences. These fences

x86		TCG IR
RMOV	→	ld;F_{rm}
WMOV	→	F_{ww};st
CMPXCHG	→	RMW
MFENCE	→	F_{sc}

(a) x86 to TCG IR

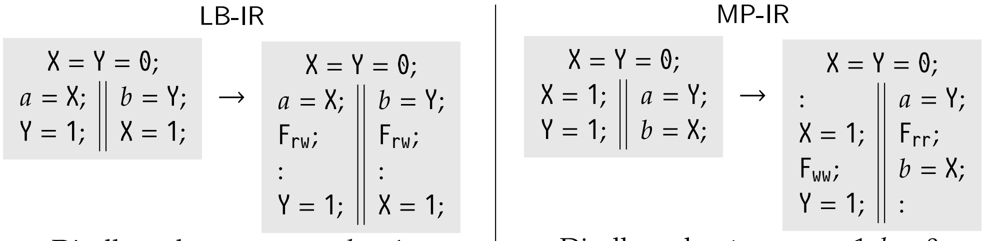
TCG IR		Arm
ld	→	LDR
st	→	STR
RMW	→	DMBFF;RMW^2;DMBFF or CAS$_{AL}$
F_{rr}/F_{rw}/F_{rm}	→	DMBLD
F_{ww}	→	DMBST
F_{wr}/F_{mm}/F_{sc}	→	DMBFF
F_{acq}/F_{rel}	→	-

(b) TCG IR to Arm

x86		TCG IR		Arm
RMOV	→	ld;F_{rm}	→	LDR;DMBLD
WMOV	→	F_{ww};st	→	DMBST;STR
CMPXCHG	→	RMW	→	DMBFF;RMW^2;DMBFF or CAS$_{AL}$
MFENCE	→	F_{sc}	→	DMBFF

(c) x86 to Arm via TCG IR

Figure 21: Verified mapping schemes for x86 to Arm via TCG IR



```
LB-IR
  X = Y = 0;                X = Y = 0;
a = X; || b = Y;    →    a = X; || b = Y;
Y = 1; || X = 1;         Frw;   || Frw;
                         :      || :
                         Y = 1; || X = 1;
```

Disallowed outcome $a = b = 1$.

```
MP-IR
  X = Y = 0;                X = Y = 0;
X = 1; || a = Y;    →    :      || a = Y;
Y = 1; || b = X;         X = 1; || Frr;
                         Fww;   || b = X;
                         Y = 1; || :
```

Disallowed outcome $a = 1, b = 0$.

Figure 22: Two examples to show the F_{rm} and F_{ww} fences in the x86→IR mappings are precise. Although weaker fences are available (*e.g.*, F_{rw} and F_{rr}), they are too weak for some programs; LB-IR shows we need at least a F_{rw} after a load, while MP-IR shows we need at least a F_{rr}, which we combine into a trailing F_{rm} for any load access in our mapping. The leading F_{ww} fence orders st-st in MP-IR. Hence we introduce a leading F_{ww} with a store access in the x86 to IR mapping.

are required[17] to preserve the mapping correctness. The mapping scheme generates a DMBLD from a $\mathtt{F}_{\mathtt{rr}}/\mathtt{F}_{\mathtt{rw}}/\mathtt{F}_{\mathtt{rm}}$ fence in the IR to preserve the order of a load with its successor memory accesses. A $\mathtt{F}_{\mathtt{wr}}/\mathtt{F}_{\mathtt{mm}}/\mathtt{F}_{\mathtt{sc}}$ fence in the IR generates a DMBFF fence to preserve the order between store-load pair on different locations. The $\mathtt{F}_{\mathtt{acq}}$ and $\mathtt{F}_{\mathtt{rel}}$ fences do not generate any instruction in Arm.

x86 to IR to Arm mapping. In Figure 21c, we combine the translations from x86 to TCG IR and from TCG IR to Arm to obtain x86 to Arm translation.

Optimizing transformations. We prove various transformations correct on concurrency primitives in TCG IR correct (Definition 3) in Agda. The verified transformations ensure the correctness of the translations in Risotto.

Memory access eliminations: TCG performs constant propagation and folding on the IR. These transformations may also be performed on shared memory accesses. Hence, we prove the correctness of the following transformations on executions, where $a \cdot b$ denotes $\mathsf{po}_{\mathsf{imm}}$-related events with the labels a and b. These are the same eliminations as those for Lasagne, explained in Section 2.6.2, given in Figure 14. We had to significantly rework their corresponding proofs from LIMM (in Chapter 2) to TIMM, as these languages provide different concurrency primitives.

Fence merging: It is correct to merge a fence to a same or stronger fence. We can also strengthen a fence to a stronger fence. We can combine these transformations as follows:

$$\mathtt{F}_{\mathtt{rm}} \cdot \mathtt{F}_{\mathtt{ww}} \xrightarrow{\text{strengthen}} \mathtt{F}_{\mathtt{sc}} \cdot \mathtt{F}_{\mathtt{sc}} \xrightarrow{\text{merge}} \mathtt{F}_{\mathtt{sc}}$$

Reordering: The plain memory accesses are unordered in TCG IR unlike in x86, and hence can be reordered freely. The proposed TCG IR model allows the reorderings of independent memory access pairs on different locations. Moreover, dependencies do not enforce any ordering in TCG IR unlike that of Arm, and hence TCG can remove false dependencies. These transformations are formally correct as the TCG IR model does not order accesses based on dependencies.

We prove that reordering $a \cdot b \rightsquigarrow b \cdot a$ is correct where a and b are the labels of non-RMW memory events which are independent and access different memory locations. The specific reorderings are the same as those for Lasagne in Section 2.6.2, given in Figure 13. Again, we had to significantly rework their corresponding proofs from LIMM (in Chapter 2) to TIMM, as they provide different concurrency primitives.

Mechanized Proofs: We prove the correctness of our transformations (Definition 3) – from some source program $\mathbb{P}^{\mathsf{s}}$ to a target program $\mathbb{P}^{\mathsf{t}}$ – in three steps in Agda. First, given a M^{t}-consistent execution X_t of $\mathbb{P}^{\mathsf{t}}$, we define a source execution X_s from $\mathbb{P}^{\mathsf{s}}$. Secondly, we relate the relations in M^{s} and M^{t} to show that X_s satisfies the axioms in M^{s}, because X_t satisfies those of M^{t}. Finally, we show that the $\mathsf{X}_t.\mathsf{mo}$ and $\mathsf{X}_s.\mathsf{mo}$

[17] Chapter 2 illustrates this in Figure 12 – these are similar here.

relations match, which means X_t and X_s have identical behaviors. The mechanized proofs consist of roughly 14,000 lines of Agda (Agda Team, 2025a).

3.7 IMPLEMENTATION

Risotto is based on QEMU 6.1.0 (QEMU Team, 2003; Bellard, 2005). In Risotto, we implement our verified mapping schemes with fast and correct translation of the x86 RMW instructions.

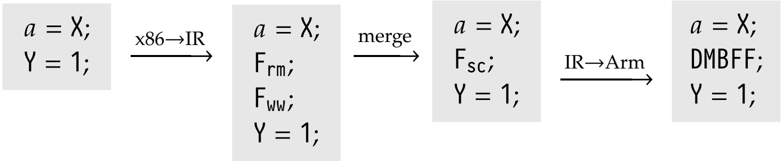
3.7.1 *Formally Verified Memory Mappings*

We implement the formally verified memory mappings from Section 3.5.2 in Risotto. More precisely, we implement the mapping schemes from Figure 21. We obtain the following performance benefits compared to the existing QEMU implementation.

Lightweight fences. Compared to QEMU that generates $\texttt{F}_{\texttt{mr}}$ and $\texttt{F}_{\texttt{mw}}$ fences before load and store operations, we generate $\texttt{F}_{\texttt{rm}}$ and $\texttt{F}_{\texttt{ww}}$ fences in the TCG IR. While QEMU's fences end up as a `DMBLD` or `DMBFF` fence, our scheme produces either a `DMBLD` or a `DMBST` fence. These fences are less costly in terms of performance than full fences Liu et al. (2020).

Newly allowed reorderings. Enforcing the proper x86 model also allows for reorderings of memory operations that were not possible with QEMU. Indeed, in our mapping scheme, there is no fence between a store and a load access. This allows store-load access pairs to be freely reordered by the processor if there is no dependency between them.

Fence merging optimizations. We implement an optimization pass over the TCG IR to merge fences that have no intermediate memory access. We merge the fences as a stronger one that suffices, and place it where the earliest fence was. As an example, we show the translation of a program from x86 to Arm: *(1)* x86 to TCG IR following Figure 21a, *(2)* fence merging, and *(3)* TCG IR to *(4)* Arm following Figure 21b.



```
a = X;
Y = 1;
```

```
a = X;
Frm;
Fww;
Y = 1;
```

```
a = X;
Fsc;
Y = 1;
```

```
a = X;
DMBFF;
Y = 1;
```

False dependency elimination. We eliminate false dependencies (*e.g.*, '`X=`a`*0` ⇝ `X=0`') on the TCG IR. It is trivially correct as the TCG IR model does not use dependency relations for any ordering, unlike in Arm.

3.7.2 *Fast and Correct CAS Instructions*

As previously detailed, QEMU translates CAS operations as calls to helper functions that in turn rely on GCC built-ins. In order to avoid the correctness problems this creates, as well as the performance degradation due to unnecessary jumps, we design a direct translation of CAS instructions. In particular, we target the translation of the x86 `CMPXCHG` instruction to Arm.

Risotto directly translates the x86 `CMPXCHG` instruction to the Arm $\texttt{CAS}_{\texttt{AL}}$ instruction, without using a helper function. We do this by adding a new instruction to the TCG IR, `CAS`. Instructions implementing a CAS semantic in the guest ISA are translated to this new TCG IR instruction if the host supports native CAS. Otherwise, the usual call to the helper function is generated. When translating back from TCG IR to the host ISA, the `CAS` instruction is translated to the corresponding host CAS instruction. More specifically, in Arm, we translate it to a $\texttt{CAS}_{\texttt{AL}}$ instruction.

Correctness. We follow the mapping schemes from Figure 21 for the RMW translation. An x86 `CMPXCHG` acts as a full fence, and only a successful $\texttt{CAS}_{\texttt{AL}}$ in Arm does the same (see Section 3.2.3). Hence, we specifically map that `CMPXCHG` instruction from x86 to Arm's $\texttt{CAS}_{\texttt{AL}}$. As both have the same ordering semantics, their translation is correct, as we have separately proven in Agda (Section 3.6).

3.8 EVALUATION

We evaluate Risotto with the Phoenix benchmarks (Ranger et al., 2007) and various PARSEC 3.0 benchmarks (Bienia, 2011). First, we observe that dynamic translation is *inherently* slower than native execution, by roughly 5–10×, which Risotto improves upon. We first translate the programs without inserting fences, to understand their effect on performance—this is clearly incorrect. After inserting the fences, we observe they account for 48% of the execution time, on average. Those results highlight the importance of reducing the overhead of fences. Our fence optimizations from Section 3.6 account for an average improvement of 6.7% over baseline QEMU. Finally, we observe that $\texttt{CAS}_{\texttt{AL}}$ outperforms baseline QEMU by 14.5% on average when there is no contention for the accessed variable. However, when increasing the number of threads and thus increasing the contention on shared variables, the performance benefits of $\texttt{CAS}_{\texttt{AL}}$ diminish. Our paper (Gouicem et al., 2022) further details Risotto's implementation and experiments[18].

3.9 CONCLUSION

We presented an end-to-end approach to provide correct and efficient execution of legacy x86 software on the weak memory Arm architecture. To achieve this, we formalize QEMU's TCG IR memory model, and use it to propose formally verified mapping schemes. We implement these schemes in Risotto, a QEMU-based DBT system

[18] As my collaborators worked on Risotto's system and experiments, I omitted those from this dissertation.

that optimizes fence placement while ensuring correctness. In contrast to our static translator Lasagne (Chapter 2), Risotto uses Arm's newer $\texttt{CAS}_{\texttt{AL}}$ instruction for fast and correct translation of x86 `CMPXCHG` instructions; whereas Lasagne required an ll/sc instruction pair surrounded by two expensive `DMBFF` fences. Through our proof mechanization, we observed Armed-Cats does not order $\texttt{CAS}_{\texttt{AL}}$ instructions strongly enough, for which we proposed a fix to the Arm model. An equivalent fix was later accepted by the Armed-Cats authors (Alglave, 2022).

Future Work. While our static translator Lasagne (Chapter 2) cannot translate all programs, as recovering semantic information statically is undecidable, this chapter's Risotto avoids that limitation by recovering the information at runtime. Unfortunately, Risotto is roughly 5–10× slower than native execution, while Lasagne was only 2–3× slower. That performance gap could be bridged by combining the strengths of both: By translating much of the program statically, while only dynamically translating the few remaining parts. We address that challenge in Chapter 4.

Finally, two of Lasagne's limitations (Section 2.9) remain: *(i)* Risotto does not consider mixed-size models (Flur et al., 2017; Alglave et al., 2021), where instructions access memory in units of multiple sizes (*e.g.*, 1/2/4/8-bytes); and *(i)* Risotto's 14,000 lines of Agda took significant proof engineering effort, which could be reduced by abstracting away domain-irrelevant mechanization detail in a general proof framework. We address the first limitation in Chapter 4 and the second in Chapter 5.

Chapter 4

Hybrid Program Translation with Mapping Schemes

ABSTRACT The current landscape of binary translation systems is fundamentally limited in terms of *completeness* for *static* systems and *performance* for *dynamic* ones. To address these limitations, we propose ARANCINI, a hybrid binary translator system designed and implemented from the ground up that strives for correct, complete, and efficient emulation for weak memory model architectures. ARANCINI makes three foundational contributions to achieve these design goals: ARANCINIR, a unified intermediate representation for static and dynamic binary translators; a formalization of ARANCINIR's memory model and formally verified mapping schemes from x86 to Arm and RISC-V, to ensure strong-on-weak correctness; and ARANCINI, a complete and performant hybrid binary translator, implementing the verified mapping schemes for correctness. To our knowledge, ARANCINI is the first hybrid binary translator whose implementation is guided by formal proofs, to ensure correct execution of strong memory guests on weak memory hosts. It is also the first translator to address mixed-sized accesses for Arm targets.

4.1 INTRODUCTION

Binary translators effectively port applications across diverse architectures, particularly in the absence of source code. Given an application binary, such translators may generate a binary for the new ISA *statically* –as seen in Chapter 2– or *dynamically* execute the *guest* binary on the new *host*—as seen in Chapter 3. Despite significant advancements, state-of-the-art binary translators suffer from challenges in *(i)* tackling the complexities of modern concurrency primitives, *(ii)* statically translating certain features without runtime information, and *(iii)* achieving good performance.

Concurrency poses subtle complexities in translating between ISAs. Architectures such as x86, Arm, and RISC-V follow different architectural rules, such as their memory hierarchy and out-of-order execution, and consequently, their memory consistency models differ. The mismatch between the consistency models may result in incorrect translation, particularly when translating from a stronger (e.g., x86) to a weaker architecture (e.g., Arm or RISC-V). Because QEMU (QEMU Team, 2003; Bellard, 2005), a state-of-the-art binary translator, sometimes translates programs *incorrectly* (as explained in Chapter 3), it has to disable concurrency altogether, severely affecting performance. To address this scenario, it is important to reason about the memory consistency models of these architectures and the translations between them.

The consistency models of x86, Arm, and RISC-V, and correct translations between weak memory architectures have been widely explored (Lustig et al., 2015;

Cota et al., 2017; Chapter 2; Chapter 3). However, the existing approaches do not suffice for emerging properties such as mixed-size concurrency (Alglave et al., 2021; Flur et al., 2017), resulting from the interplay between concurrency and *mixed-size* accesses (*e.g.*, the Linux kernel's `lockref` structure explored by Alglave et al. (2021) and Flur et al. (2017)). Hence, it is imperative to overhaul the reasoning about translation correctness under the mixed-size concurrency models of architectures.

Besides correctness, existing binary translators face limitations in completeness and performance. SBTs (Fu et al., 2018; Guo et al., 2016), including Lasagne (Chapter 2), bring performance but lack completeness as they cannot translate all features ahead-of-time since static disassembly in itself is undecidable (Rice, 1953). In contrast, DBTs (QEMU Team, 2003; Spink et al., 2019), including Risotto (Chapter 3), are complete but come with a significant performance penalty due to both runtime compilation latency, and reduced optimization opportunities.

Modern hybrid systems (Shen et al., 2012b; Yang et al., 2024; Wang et al., 2019; Microsoft, 2024), such as Apple's Rosetta (Apple Inc., 2021), combine both to achieve complete and performant translation, but do not provide any correctness guarantees, require proprietary hardware to cope with the different memory-models (Jha, 2020), or simply do not allow strong-on-weak translation.

To address these challenges, we propose Arancini, an end-to-end Hybrid Binary Translation (HBT) system, designed and implemented from the ground up, for correct and efficient *strong-on-weak* binary translation with mixed-size accesses.

We address the correctness challenges by defining a weak memory model for Arancini's intermediate language ArancinIR, using which we prove memory access mappings correct, from the strong guest to the weak host models. In particular, our mappings correctly translate mixed-size accesses (Flur et al., 2017; Alglave et al., 2021) from x86 to ArancinIR to Arm. We implement those correct mappings in Arancini.

Mixed-size extensions to weak-memory models introduce subtle complexities in program behavior and translation correctness (Marmanis et al., 2025; Sato et al., 2025). Although we developed mapping schemes from x86 to Arm in Chapters 2 and 3, those did not consider mixed-size concurrency models. The mapping schemes from x86 to Arm in this chapter address that challenge, which required extensive proof mechanization efforts. Moreover, we discovered common access-splitting optimizations are incorrect under mixed-size concurrency. Finally, translating to RISC-V with mechanized proofs is a new contribution over those earlier chapters.

We implement Arancini with an x86 frontend and two backends: Arm and RISC-V. Our verified translation rules *correctly* enforce the x86 ordering on Arm and RISC-V, even for subtle mixed-size accesses (on Arm).

Contributions. Our foundational contribution is a **formal memory model for our ArancinIR and correctness proofs of memory mappings** between x86, ArancinIR, Arm and RISC-V, mechanized in the Agda proof assistant (Agda Team, 2025a). These translation rules direct the correctness of the translations (Section 4.4).

4.2 BACKGROUND AND MOTIVATION

4.2.1 *Weak Memory Concurrency*

In Chapter 1 we already covered the general challenges of weak memory concurrency. Here, we expand on a recent feature, being *mixed-size* concurrency.

Mixed-size concurrency. Recently, memory models started supporting *mixed-size* accesses in units of multiple sizes (*e.g.*, 1/2/4/8-byte units) (Flur et al., 2017; Alglave et al., 2021). For example, `lockref` and read-copy-update (RCU) implementations in the Linux kernel, Arm v8.0 ticketed spinlock, and the implementation of FreeBSD/i386 PAE page tables perform mixed-size accesses in concurrent programs. In addition, state-of-the-art architectures such as x86 and Arm provide instructions to perform mixed-size accesses to optimize programs. These mixed-size accesses affect concurrency behaviors, as shown in the following example:

$\mathtt{Y}_L$ = 0x01 t = Y;	Y = 0x0203; (Mixed)

`Y` is a 16-bit variable shared by the two concurrently running threads, initially `0`. $\mathtt{Y}_L$ and $\mathtt{Y}_H$ denote the least and most significant byte, respectively. The outcome `Y=0x0203`, t`=0x0201` is forbidden in both x86 and Arm.

Correctness of binary translation. Ignoring the differences in weak memory models across architectures can cause errors when translating between them. Correctly translating concurrent programs demands paying close attention to these formal memory models. However, unlike our earlier approaches (Chapters 2 and 3), we also need to reason about the mixed-size concurrency models to correctly translate mixed-size accesses. Finally, while Lasagne and Risotto translated only to Arm, we now also target RISC-V, demanding separate formal examination.

4.2.2 *Static Binary Translation*

Binary translators traditionally classify as either *static*, translating binary programs *ahead of time*, or *dynamic*, translating them *at runtime*. Each approach offers unique strengths, which we incorporate in our solution.

Since a Static Binary Translation (SBT) translates programs before execution, it can spend some time optimizing the code before generating the final program. To do so, state-of-the-art SBTs often *lift* binary programs to a higher-level IR (Bougacha, 2022; Trail of Bits, 2022; Yadavalli and Smith, 2019; Avast Developers, 2022; Shen et al., 2012a), such as LLVM IR (LLVM Team, 2025; Lattner and Adve, 2004). After lifting, these programs benefit from numerous optimizations implemented in LLVM. Additionally, as LLVM is a *compiler* infrastructure, it naturally generates code for the target architecture. As LLVM handles much of the complexity, the primary challenge in static translation is lifting the source binary program to an IR.

Completeness of binary translation. SBTs are *necessarily* incomplete (Rice, 1953; Andriesse et al., 2016), meaning they cannot translate all programs. When compiling a program written in a high-level language, much information is lost. In particular, control flow structures and jump targets are removed. Reconstructing that information from a binary is difficult in practice and *impossible* in general (Rice, 1953). Additionally, lifting binary programs brings issues that do not exist in the source language, such as distinguishing between pointers and constants, or between data and instruction bytes (Andriesse et al., 2016). Consequently, state-of-the-art SBTs (Trail of Bits, 2022; Kim et al., 2025) –including our Lasagne (Chapter 2)– fail to lift many binary programs.

4.2.3 *Dynamic Binary Translation*

In contrast, a DBT can be complete, since it only translates program fragments at runtime within their local execution contexts. For instance, the state-of-the-art DBT QEMU (QEMU Team, 2003; Bellard, 2005) translates a binary program at basic-block granularity to its TCG IR QEMU Team (2025). State-of-the-art DBTs (QEMU Team, 2003; Spink et al., 2019; Hong et al., 2012; Cota et al., 2017) –including our Risotto (Chapter 3)– distinguish themselves by emulating different fragments of the host machine, such as different RMW instructions and floating-point operations.

TCG IR. Static and dynamic translators operate at different levels of abstraction, and thus their IRs embody different design decisions. While LLVM IR was designed to compile high-level languages, where control flow is known, TCG operates on smaller local fragments together with runtime information.

Performance of binary translation. Unfortunately, while local translation ensures completeness, it lacks a global program view, preventing many optimizations. To showcase the performance gap between DBT and SBT, we compare our[19] cross-ISA binary translators, Lasagne (Rocha et al., 2022) and Risotto (Gouicem et al., 2022), against native host binaries for the Phoenix (Ranger et al., 2007) benchmark suite, which both translators use in their evaluation.

While Risotto is 5–10× slower than native, Lasagne manages to generate binaries that are only 2–3× slower. Lasagne performs so well for two reasons: *(i)* it relies on the LLVM compiler and its function-level optimizations, and *(ii)* it doesn't need to fully emulate the x86 CPU state (unlike DBTs). This performance overhead makes DBTs unappealing, it's completeness is valued more in practice. In the current landscape of binary translation, one has to choose between *completeness* and *performance*.

[19] Although Chapters 2 and 3 present the formal memory models and mappings of Lasagne and Risotto, their corresponding publications include the experimental results – as those were not *my* personal contribution.

4.2.4 *Proposal: A Correct Hybrid Binary Translator*

While formal reasoning over memory models grants *correctness*, and DBT grants *completeness*, we can take advantage of SBT as much as possible to gain *performance*. Hence, to address performance, completeness, and correctness, we propose an end-to-end hybrid binary translation system that: *(i)* statically generates performant binaries; *(ii)* provides dynamic capabilities to achieve completeness; and *(iii)* relies on formally verified translation rules for concurrent programs.

4.3 OVERVIEW

In this section, we present Arancini, an end-to-end hybrid binary translation system that can translate a binary compiled for one ISA into an equivalent binary that will run on another ISA, especially targeting weak memory model architectures. Arancini statically lifts as much of the program as possible, like Lasagne (Chapter 2). However, unlike Lasagne, which aborts upon encountering fragments it cannot lift statically, Arancini translates those dynamically, like Risotto (Chapter 3).

4.3.1 *Design Challenges and Key Ideas*

Combining static and dynamic translation presents several challenges, arising from their conflicting requirements. We identify three core challenges that must be overcome to effectively implement a hybrid translation scheme.

Challenge #1: Bridging the gap between static and dynamic binary translation. IRs like TCG IR or LLVM IR, used by different translators, are designed for a single translation scheme only. While TCG IR is low-level and designed to efficiently encode guest hardware state, its small translation units prohibit global optimizations. LLVM IR provides high-level constructs, such as functions, to perform such optimizations. However, the aggressive optimizations that it can perform are often time-consuming and memory-heavy, making it too slow for use in dynamic translation.

We design ArancinIR to facilitate both high-performance static translation and low-overhead dynamic translation, using a single frontend to reduce translation mismatches.

Challenge #2: Ensuring architectural correctness. Any translation system needs to honor the semantics of the guest architecture, in the generated translated code. We particularly consider the memory ordering semantics, which can be vastly different between two architectures, as we already explained in Chapters 1 to 3. However, we must again now the translation from x86 to ArancinIR to Arm, which differs from our mapping schemes in Chapter 1, which used LIMM as intermediate model; and Chapter 3, which used TIMM as intermediate model. In addition, those prior mappings do not give any formal guarantees for mixed-size accesses, which is a primary challenge we address in this chapter.

x86		ArancinIR		Arm	RISC-V
RMOV	→	ld; F_{rm}	→	LDR; DMBLD	LD; F_{rm}
WMOV	→	F_{ww}; st	→	DMBST; ST	F_{ww}; ST
CMPXCHG	→	RMW	→	CAS_{AL}	RMW_{sc}
MFENCE	→	F_{mm}	→	DMBFF	F_{mm}
		F_{rr}/F_{rw}	→	DMBLD	F_{rr}/F_{rw}
		F_{wr}/F_{wm}/ F_{mr}/F_{mw}	→	DMBFF	F_{wr}/F_{wm}/ F_{mr}/F_{mw}

Figure 23: Verified mapping from x86 to Arm and RISC-V via ArancinIR.

4.4 AIMM WEAK MEMORY MODEL AND MAPPING SCHEMES

In this section, we propose the formal ArancinIR memory model (AIMM). In particular, we prove *mapping schemes* between those architectures correct, using existing architecture semantics that describe their weak memory behavior. We show those resulting mapping schemes in Figure 23, whose implementation in Arancini *formally guarantees* correct translation of weak memory behaviors from x86 to Arm and RISC-V. Specifically, our mapping restricts the weak behaviors on Arm and RISC-V to those observable on x86. In this section, we formally define AIMM and give a high-level explanation of our proofs for those mappings, which we mechanized in the Agda proof assistant (Agda Team, 2025a).

4.4.1 *Mixed-Size Accesses*

We build upon the semantics introduced in Chapter 1, but extended with another *primitive* “same instruction” relations si to capture semantics for mixed-size accesses (Flur et al., 2017; Alglave et al., 2021). In the prior non-mixed models, every load instruction generates *one* R event and every store instruction generates *one* W event. In mixed-size models, either instruction can generate multiple events of the same type. For instance, a 16-bit load generates *two* R events, one for each byte read, connected by a si edge; similarly, a 32-bit store generates *four* W events, all connected by a si edges. Hence, si is an *equivalence*.

We use existing mixed-size x86 and Arm models from Alglave et al. (2021), where latter uses updated definitions by Alglave and Maranget (2025), particularly those including the CAS_{AL} fix (Alglave, 2022) (from Section 1.3.3). Note that RISC-V does not currently have a formal mixed-size model.

Example. Consider the Mixed execution in Figure 24. Contrary to models *without* mixed-size accesses, the program order (po) is *not* total per thread, as si-related events occur simultaneously in po. Observe that accessing 16-bit location Y produces two si-related 8-bit accesses. The outcome is disallowed by the rfe;si;fre;si cycle in all models. However, observe that splitting either 16-bit access instructions of Y into two 8-bit

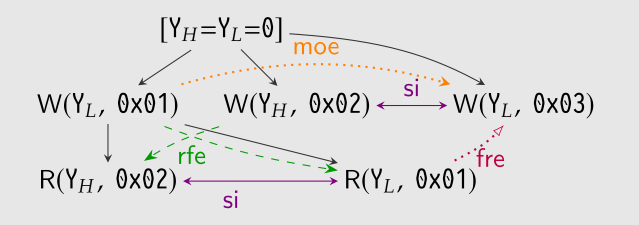


Figure 24: Execution of the Mixed program, with outcome Y=0x0203,t=0x0201, which is forbidden in x86 and Arm. Observe that, without the si edges, there are no cycles – which means the executions would be allowed if we had not modeled si edges.

accesses (to Y_H and Y_L) discards the corresponding si edge, thus breaking the cycle and allowing the outcome.

4.4.2 *Concurrency Model: AIMM*

We define the Arancini concurrency primitives and their corresponding generated ArancinIR events and relations in an execution.

Memory accesses. Arancini provides load (ld) and store (st) operations that respectively read from and write to shared memory locations. ld and st accesses generate R and W events. It also provides an atomic read-modify-write (RMW) operation, which orders with surrounding events, like x86's CMPXCHG, Arm's $\mathsf{CAS}_{\mathsf{AL}}$, and RISC-V's $\mathsf{RMW}_{\mathsf{sc}}$. To ensure it also orders upon failure, we define the ArancinIR to generate a distinct R_{SC} event and, if successful, a subsequent rmw-related W_{SC} event.

Fence accesses. Arancini provides different types of fences: F_{rr}, F_{rw}, F_{rm}, F_{wr}, F_{ww}, F_{wm}, F_{mr}, F_{mw}, and F_{mm}. These fences generate F_{RR}, F_{RW}, F_{RM}, F_{WR}, F_{WW}, F_{WM}, F_{MR}, F_{MW}, F_{MM} events, respectively. All these fences order certain memory accesses which we capture in *order* (ord) relations. For instance, a pair of po-related events (a, b) are in ord if a and b are R events with an intermediate F_{RR} event, which follows the [R];po;[F_{RR}];po; [R] case of ord.

Relations. Next, to capture mixed-size accesses, we compose rfe, moe, and fre relations with the si relation, denoted by rfe;si, moe;si, fre;si respectively. For example, if $(w, r) \in$ rfe and $(r, e) \in$ si then w and e are ordered by rfe;si, which can happen when reading 16 bits from two distinct 8-bit stores. Finally, we define *global-happens-before* (ghb) order using ord, and rfe;si, moe;si, fre;si relations to order events across different threads.

Axioms. Similarly to all the architectures, our AIMM model also satisfies the Coherence and Atomicity axioms (from Chapter 1). In addition, the GOrd axiom below ensures acyclicity of the ghb order in an AIMM execution.

$$\mathsf{ghb} \text{ is irreflexive} \qquad \text{(GOrd)}$$

$$\begin{array}{rlll}
\text{where } \mathsf{ghb} \triangleq & \multicolumn{3}{l}{(\mathsf{ord} \ \cup \ (\mathsf{rfe};\mathsf{si}) \ \cup \ (\mathsf{moe};\mathsf{si}) \ \cup \ (\mathsf{fre};\mathsf{si}))^+} \\
\mathsf{ord} \ \triangleq & & [\mathsf{R}];\mathsf{po};[\mathsf{F}_{\mathsf{RR}}];\mathsf{po};[\mathsf{R}] & \cup \ [\mathsf{R}];\mathsf{po};[\mathsf{F}_{\mathsf{RW}}];\mathsf{po};[\mathsf{W}] \\
& \cup & [\mathsf{R}];\mathsf{po};[\mathsf{F}_{\mathsf{RM}}];\mathsf{po};[\mathsf{R} \cup \mathsf{W}] & \cup \ [\mathsf{W}];\mathsf{po};[\mathsf{F}_{\mathsf{WR}}];\mathsf{po};[\mathsf{R}] \\
& \cup & [\mathsf{W}];\mathsf{po};[\mathsf{F}_{\mathsf{WW}}];\mathsf{po};[\mathsf{W}] & \cup \ [\mathsf{W}];\mathsf{po};[\mathsf{F}_{\mathsf{WM}}];\mathsf{po};[\mathsf{R} \cup \mathsf{W}] \\
& \cup & [\mathsf{R} \cup \mathsf{W}];\mathsf{po};[\mathsf{F}_{\mathsf{MR}}];\mathsf{po};[\mathsf{R}] & \cup \ [\mathsf{R} \cup \mathsf{W}];\mathsf{po};[\mathsf{F}_{\mathsf{MW}}];\mathsf{po};[\mathsf{W}] \\
& \cup & [\mathsf{R} \cup \mathsf{W}];\mathsf{po};[\mathsf{F}_{\mathsf{MM}}];\mathsf{po};[\mathsf{R} \cup \mathsf{W}] & \cup \ [\mathsf{codom}(\mathsf{rmw})];\mathsf{po};[\mathsf{R} \cup \mathsf{W}] \\
& \cup & [\mathsf{R}_{\mathsf{SC}}];\mathsf{po};[\mathsf{R} \cup \mathsf{W}] & \cup \ [\mathsf{R} \cup \mathsf{W}];\mathsf{po};[\mathsf{W}_{\mathsf{SC}}]
\end{array}$$

The ord relations captures the preserved thread-local orders between instructions, like xlob in x86, ob in Arm, and rppo in RISC-V (from Section 1.3). Notably, Arancini's *nine* fences give fine-grained control over ordering behavior, unlike Arm. In addition, RMW-generated R_{SC} and W_{SC} events order with many surrounding events – together with the rule with 'codom(rmw)', those rules ensure a rmw orders like a full fence. Finally, because ghb includes the si edges, the Mixed execution in Figure 24 is disallowed in AIMM, like in x86 and Arm – it has a '(rfe;si);(fre;si)' cycle.

4.4.3 *Mappings and Optimizations*

We define the mapping schemes in Figure 23 and prove them correct.

Correct mapping schemes. We translate concurrency primitives from x86 to Arm and RISC-V in two steps: *(1)* x86 to ArancinIR and *(2)* ArancinIR to Arm/RISC-V. We prove them correct (Definition 3 in Section 1.4) in Agda (Agda Team, 2025a). The mappings are also *precise* (Definition 4), meaning each fence in the mapping is needed for *at least one* program.

Figure 23 also contains the full composed mappings, combining their respective components. We need not explicitly prove correctness (Definition 3) again, as the components trivially compose. From x86 to Arm, the guarantees extend to mixed-size accesses. However, as RISC-V does not have a mixed-size model, the proofs of its corresponding mappings do not consider them either. As a consequence, the composed x86→AIMM→RISC-V mapping does not guarantee correctness of mixed-size accesses.

Optimizations. Although ArancinIR allows optimizations (like LIMM and TIMM in Section 2.6.2 and 3.6), through our proof efforts we observed that *splitting* memory accesses is invalid. For instance, it is incorrect to split a 16-bit store to X into two separate 8-bit stores to its most-significant X_H and least-significant X_L byte (with X initially 0):

$$\begin{array}{l||l} \mathtt{X} \overset{16}{=} \mathtt{0x1234;} & a \overset{8}{=} \mathtt{X}_L\mathtt{;} \\ & \mathtt{F_{rm};} \\ & \mathtt{X}_H \overset{8}{=} \mathtt{0xFF;} \end{array} \quad \not\rightarrow \quad \begin{array}{l||l} \mathtt{X}_L \overset{8}{=} \mathtt{0x34;} & a \overset{8}{=} \mathtt{X}_L\mathtt{;} \\ \mathtt{X}_H \overset{8}{=} \mathtt{0x12;} & \mathtt{F_{rm};} \\ & \mathtt{X}_H \overset{8}{=} \mathtt{0xFF;} \end{array}$$

In the original program (left), producing `X = 0x1234` at the end means that a = `0` because the right thread completed before the left starts. However, by splitting the 16-bit store to `X`, we *incorrectly* introduce the additional outcome `X=0x1234`, a=`0x34`. This is not merely a consequence of AIMM's decisions, as splitting accesses is also incorrect in x86, Arm, and RISC-V. In general, correctly splitting accesses requires globally locking the split instructions.

4.4.4 *Proofs*

We mechanize all our proofs in Agda (Agda Team, 2025a). Regarding mixed-size accesses, we model the si relation as an *equivalence* relation between a pair of R-R or W-W accesses. Our particular proof mechanization challenges include:

- The definitions of the mixed and non-mixed architecture models for x86 and Arm are structured very differently, making our novel mechanized proofs very different from our previous non-mixed proofs between x86 and Arm (Chapters 2 and 3).
- As the RISC-V model differs significantly from Arm, it required additional proof efforts.
- Modeling the si relation and related lemmas affects significant parts of the other proof components.

4.5 EVALUATION

We evaluate[20] Arancini around three dimensions: performance, completeness, and correctness. First, we observe that the programs we translated with Arancini are, on average, 8.01× slower than native for Arm and 4.52× slower for RISC-V. In particular, we observe Arancini often performs better than our dynamic translator Risotto while sometimes performing similarly. Second, we observe that Arancini is more *complete* than our static translator Lasagne, as it can translate all programs in the Phoenix benchmark suite (Ranger et al., 2007), which is two more than Lasagne. Finally, we note that both Lasagne and Risotto did not provide any formal guarantees for translating mixed-size accesses, which Arancini does with its mapping schemes proved over the newer mixed-size models.

[20] As my collaborators worked on Arancini's system and experiments, I omitted them from this dissertation.

4.6 CONCLUSION

In this chapter we presented our *hybrid* binary translator Arancini. Contrasting our prior static translator Lasagne (Chapter 2), which cannot lift all programs, Arancini seamlessly switches to dynamic translation whenever static translation is impossible (*e.g.,* for dynamic jump targets). However, contrasting our dynamic translator Risotto (Chapter 3), the hybrid approach translates many program fragments *before* executing the program, benefiting performance.

Most notably, we defined another memory model AIMM for Arancini which captures the subtle semantics of mixed-size accesses, unlike LIMM (Chapter 2) and TIMM (Chapter 3). Consequently, Arancini guarantees correct translation of weak memory behaviors from x86 to Arm through AIMM. In addition, unlike our prior mapping schemes, we defined and proved a mapping scheme to RISC-V.

Future Work. A primary limitation of the proofs of Lasagne (Chapter 2) and Risotto (Chapter 3) remains with Arancini: They are a large and rigid code base. In Chapter 5 we address that limitation by defining a general proof framework for mechanizing axiomatic weak memory proofs in Agda, allowing future proof engineers to ignore domain-irrelevant mechanization details when writing such proofs.

With Arancini we improved performance over dynamic translation (Chapter 3), getting closer to static translation (Chapter 2). However, binary translation has inherent overheads, largely originating from the compilation process that produced the x86 binary, where information was lost that cannot fully be reconstructed. Instead, we could take an alternative approach from binary translation when: *(i)* we have the program source code, and *(ii)* we have time to thoroughly analyze the program. Under those conditions we expect to insert significantly fewer fences, which we explore in Chapter 6.

Binary Translation Related Work

Here we give the related work shared by Chapters 2 to 4, primarily on concurrency semantics and binary translation.

CONCURRENCY SEMANTICS

High-Level Languages. Compiling concurrency primitives in programming languages is well-studied by Batty et al. (2012); Sarkar et al. (2012); Alglave et al. (2014); Petri et al. (2015); Podkopaev et al. (2019). However, compiling a program written in a high-level language (*e.g.,* Java or C/C++) to low-level machine code differs from our context of translating between low-level languages in two primary ways:

- Weak memory models of high-level languages often contain elaborate inter-thread synchronization sequences (Batty et al., 2011; Lahav et al., 2017), while the architecture models we consider capture reordering locally per thread.
- Data races result in undefined behavior in C/C++ (Batty et al., 2011; Chakraborty and Vafeiadis, 2017, 2019; Kang et al., 2017; Lahav et al., 2017), while in LLVM a read-write race has defined behavior where the racy read returns `undef` and write-write races result in undefined behavior (Chakraborty and Vafeiadis, 2017). In contrast, the architecture models we consider do not have undefined behavior.

Hence, those existing language models and their corresponding compilation correctness results do not apply when translating between architectures. Our models are closer to hardware memory models and have no undefined behavior.

Intermediate Languages. Similar to our LIMM (Chapter 2), TIMM (Chapter 3), and AIMM (Chapter 4) memory models, which are intermediate between architectures, Podkopaev et al. (2019) propose an "*intermediate memory model*" IMM, which is intermediate between high-level languages and architectures. As IMM is primarily used to compile C++ to various architectures, it captures semantics for many C++ concurrency primitives (Lahav and Margalit, 2019), while mappings from promising semantics (Kang et al., 2017) to IMM are also proven correct. Finally, they proved mappings to various architectures, including x86-TSO, Armv8, and RISC-V, which we also consider.

Unlike our models, IMM orders memory accesses with dependencies, which existing LLVM optimizations do no preserve. As a consequence, adapting IMM to our context while preserving correct compilation requires to either *(i)* restrict or disable those optimizations, which would reduce performance of the resulting program; or to *(ii)* explicitly check IMM-specified dependencies during compilation and adapt existing optimizations to use that information, which would require extensive compiler modifications. Instead, we explicitly developed our models to enforce no such ordering, meaning we can safely apply existing optimizations.

Transformations. Program transformations under weak memory models have been studied extensively. Morisset et al. (2013); Chakraborty and Vafeiadis (2016); Vafeiadis et al. (2015) have identified errors in existing C++ compilers, while Ševčík and Aspinall (2008) identified errors when compiling Java. Kang et al. (2017); Lee et al. (2020); Vafeiadis et al. (2015) propose general memory models on which common optimizations are correct, as we do for binary translation with LIMM, TIMM, and AIMM. Ševčík (2011) proves common transformations correct for a C++-like model, as we do for our models.

Differences between Memory Models. Similar to our examination of weak memory model differences between x86, Arm, and RISC-V, others have also studied differences between weak memory models. Adve and Hill (1993); Adve and Gharachorloo (1996); Higham et al. (1997); Steinke and Nutt (2004); Higham et al. (2007); Alglave (2012) introduce formal frameworks to express characteristics of different memory models, while Mador-Haim et al. (2010); Alglave et al. (2010); Wickerson et al. (2017) propose automatic approaches to identify differences between models. In contrast, we developed intermediate formal memory models that are intermediate between architectures. We used a proof assistant to establish that behavior is preserved when translating between architectures with our mapping schemes.

Similar to how our mapping schemes enforce a strong model on a weaker architecture, Shasha and Snir (1988); Sura et al. (2005); Lee and Padua (2001) propose program analysis algorithms that identify where fences (and synchronization) must be *inserted* to enforce a stronger model. Vafeiadis and Zappa Nardelli (2011); Elhorst (2014); Morisset and Nardelli (2017) propose algorithms to *eliminate* excessive fences; as we do with our local fence elimination and merging rules. Unlike those other approaches, our context of (dynamic) binary translation has little runtime computation budget for program analyses. Our mapping schemes thus always insert fences with memory accesses, while we separately define local fence optimizations. Although our approach does not minimize the total number of fences, it works well in our resource-constrained settings. Regardless, any approach—even with unlimited computation budget—would insert too many fences, in general, as optimal fence placement is undecidable (Atig et al., 2012).

Robustness Enforcement. Robustness (or stability) based approaches check, by exploring executions, if a given program is SC-robust against weaker models, inserting fences otherwise (Bouajjani et al., 2013a; Liu et al., 2012; Alglave et al., 2017; Lee and Padua, 2001; Shasha and Snir, 1988; Linden and Wolper, 2011, 2013; Abdulla et al., 2015b; Margalit and Lahav, 2021; Lahav and Margalit, 2019; Oberhauser et al., 2021; Chakraborty, 2021). In contrast, our schemes require no program analysis, *e.g.*, model checking, as they are correct for any program. While our mappings often insert more fences than needed, they are easily applied to large programs, unlike expensive analysis approaches.

Recently, Beck et al. (2023) proposed AtoMig to migrate large programs from x86 to Arm, like our binary translators. AtoMig analyzes LLVM IR programs to identify x86-TSO robustness violations on Arm. As it has access to program source code, it preserves more of the original program written in a high-level language, making its produced programs perform better than our LASAGNE's (Beck et al., 2023, §4.3). However, unlike our mapping schemes, AtoMig is *not* formally correct, meaning it may insert too few fences.

BINARY TRANSLATION

Static Binary Translation. Static translators translate instructions one-by-one from one ISA to another (Andrews and Sand, 1992; Sites et al., 1993). More recently, several static translators have adopted intermediate representations (Bougacha, 2022; Trail of Bits, 2022; Yadavalli and Smith, 2019; Avast Developers, 2022; Shen et al., 2012a), enabling the lifted code to be further optimized and more easily compiled to different targets. These existing works cannot translate concurrent programs from strong to weak architectures.

Dynamic Binary Translation. Many dynamic binary translators precede our RISOTTO, such as those by Cota et al. (2017); Guo et al. (2016); Hong et al. (2012); Wang et al. (2011); Ding et al. (2011); Lustig et al. (2015). Particularly, Rosetta 2 is a commercial tool for Apple Silicon (Apple Inc., 2021) that translates x86 programs to the Arm ISA. However, Apple Silicon implement both Arm *and* x86-TSO models in hardware. Rosetta 2 thus handles the memory model mismatch by enabling the latter model when translating legacy x86 programs (Jha, 2020). In contrast, our mapping schemes are entirely software-based.

Various QEMU-based DBTs emulate multi-threaded programs. Most of them do not address mismatches among memory consistency models (Ding et al., 2011; Wang et al., 2011; Hong et al., 2012). Lustig et al. (2015) propose ArMOR where a specification format defines the ordering of memory accesses in architectures, allowing it to identify the required fences during a program execution. However, they do not handle the subtle semantics of `RMW` accesses and dependency based orderings. We handle these features while reasoning about the translation rules. Moreover, for x86 to Armv8 translation, ArMOR uses release-acquire accesses and enforces SC semantics on Arm. Pico (Cota et al., 2017) follows ArMOR rules in its mappings, but does not provide any formal guarantee of correctness. Moreover, they focus on translation from PowerPC to x86, which is different from our schemes.

Chapter 5

Weak Memory Mapping Proofs in Agda

ABSTRACT Writing correct transformations for concurrent programs on weak memory architectures is challenging, particularly because architectures show various weak behaviors. To ensure those transformations are *correct*, one must prove them correct by thoroughly inspecting the subtle weak memory semantics of the relevant architectures. Writing such proofs is often time-consuming and prone to errors.

To address this challenge, we propose the *domain-specific* proof framework BURROW to mechanize axiomatic weak memory semantics and prove the corresponding transformations correct in the Agda proof assistant.

First, we identify challenges in mechanizing weak memory proofs, which are often overlooked in pen-and-paper proofs. Second, we explain our domain-specific proof primitives, with which we prove large fragments of weak memory mapping proofs *generally*, allowing programmers to focus only on the interesting parts of their proofs. Finally, we use BURROW to prove a conjectured mapping from x86 to Arm.

5.1 INTRODUCTION

Formal proofs can that show transformations in weak memory models (Alglave et al., 2014, 2021) are correct. These proofs are complex, involving many subtle cases, and are often error-prone (Sarkar et al., 2011, 2012; Batty et al., 2016; Manerkar et al., 2016). To avoid missing cases, we can *mechanize* these proofs in a *proof assistant*, such as Agda (Agda Team, 2025a), to significantly increase their reliability.

Existing approaches (Podkopaev et al., 2019; Alglave et al., 2014; Batty et al., 2011; Doko and Vafeiadis, 2016) often bring significant theoretical contributions. For instance, by finding errors in existing models (Chakraborty and Vafeiadis, 2016, 2017), repairing those models (Lahav et al., 2017; Chapter 3), or introducing altogether new models (Podkopaev et al., 2019; Batty et al., 2011; Kang et al., 2017). However, *mechanizing* such theoretical results in a proof assistant is an *orthogonal* problem, which few solutions address. Primarily, IMM (Podkopaev et al., 2019) proposes an intermediate memory model for compiler correctness proofs, accompanied by significant mechanized proofs in Rocq (Bertot and Castran, 2010).

A remaining issue is that *writing* mechanized proofs is time-consuming, as was the case for our proofs in Chapters 2 to 4. Proof assistants simplify mechanization in various ways; for instance, idiomatic Rocq favors extensive use of *tactics* to automatically solve goals. In contrast, Agda's explicit term manipulation with dependent pattern matching and easy syntax extensions – but lack of tactics – emphasizes using the right formalization abstractions (Agda Team, 2025c). While proving the mapping schemes in the previous chapters, we discovered commonalities in their proof structures, which benefit from abstractions to reduce their proof complexity.

We propose the *domain-specific* proof framework BURROW, to mechanize weak memory transformation proofs in Agda. BURROW includes primitives and abstractions, addressing formalization patterns commonly appearing when *proving* axiomatic weak memory transformations *by contradiction*. BURROW greatly reduces the mechanization effort and "boilerplate code" needed for weak memory transformation proofs.

Contributions. We make the following contributions:

- **Clarify Weak Memory Definitions** – Although weak memory semantics (Alglave et al., 2014) are formal models, many of its definitions are given only *implicitly*; particularly, various definitions needed for proof mechanization are missing. While pen-and-paper proofs may permit unspecific definitions, mechanized proofs demand meticulous specifications. We identify and clarify several such definitions (Section 5.3).
- **Relation Proof Library** DODO – Axiomatic weak memory semantics are defined over predicates and binary relations. While Agda's standard library (Agda Team, 2025b) includes many relevant definitions, many more are needed for weak memory proofs. As those definitions are *not specific to* weak memory, we develop these *from the ground up* in a proof library DODO.
- **Transformation Proof Framework** BURROW – We identify several *domain-specific* abstractions for weak memory transformations proofs, while also proving large fragments *generally*. We develop, *from the ground up*, the Agda proof framework BURROW for weak memory transformation proofs (Section 5.4).
- **Case Study: x86⇒Arm Mapping Proof** – We demonstrate BURROW's effectiveness by proving a conjectured mapping from x86 to Arm correct (Alglave et al., 2021). Crucially, the mapping was originally incorrect –as we explained in Chapter 3– but would be correct after fixing the model (Alglave, 2022). To the best of our knowledge, we are the first to prove it correct under that fixed Arm model (Section 5.5).

5.2 BACKGROUND

We have already thoroughly explained the intricacies of weak memory semantics in Chapter 1. In this chapter, we build on those same models and definitions; we start by looking at the structure of the theorems we prove with BURROW.

5.2.1 *High-Level Theorem Statement*

To ensure that a weak memory transformation is correct, we must guarantee that it introduces no additional behavior. In Chapter 1 we saw that programs may generate any of multiple executions. From those intuitions we can visualize the correctness criterion as follows (corresponding to Definition 3 from Section 1.4):

$$\begin{tikzcd} \mathbb{P}^s \arrow[r, "\text{map}"] \arrow[d, "\text{execute}"'] & \mathbb{P}^t \arrow[d, "\text{execute}"] \\ G^s \arrow[r, phantom, "\supseteq_{\mathsf{behavior}}"] & G^t \end{tikzcd}$$

The source program $\mathbb{P}^s$ executes under the source memory model, producing *a set* of executions G^s. The source program's syntax 'map's to a target program $\mathbb{P}^t$. $\mathbb{P}^t$ executes under the target model, also producing *a set* G^t. For each graph set, we consider its set of behaviors (from Definition 2); as the transformation *must not* introduce new behaviors, we require '$\{ \mathsf{behavior}(X^t) \mid X^t \in G^t \} \subseteq \{ \mathsf{behavior}(X^s) \mid X^s \in G^s \}$'.

Program 1: Mechanized Theorem Statement (Agda)

```
proof :
  -- Given a target execution
  {dst : Execution}
  -- Which is well-formed
  (dst-wf : WellFormed dst)
  -- Which is consistent with the target language's memory model
  (dst-ok : IsDstConsistent dst)
  -- and produced by a program whose syntax was mapped by the mapping scheme
  (dst-res : MappingRestricted dst)
  → -- We produce a source execution
    ∃[ src ]
      ( -- Which is well-formed
        WellFormed src
      × -- Which is consistent in the source language's memory model
        IsSrcConsistent src
      × -- and produced by the source program mapped to produce `dst`
        SyntaxMapped src dst
      × -- Terminating with the same memory as `src`
        behavior src ⇔₂ behavior dst)
```

We selected Agda (Agda Team, 2025a) as our proof assistant, within which we mechanize the high-level theorem statement in Program 1. We aim to keep our Agda notation close to mathematical notation. Agda is a dependently-typed functional programming language, where types are theorems. We prove a weak memory mapping correct by finding an inhabitant of `proof`. For each particular target execution $X^t \in G^t$, we *produce* a source execution $X^s \in G^s$. In the subsequent Sections 5.3 and 5.4 we expand on the definitions, but give an overview here:

- `WellFormed` – Well-formed executions "make sense", which is formally captured with properties about events and primitive relations (*e.g.*, $\mathsf{rf} \subseteq \mathsf{W}{\times}\mathsf{R}$, $\mathsf{rmw} \subseteq \mathsf{po}$, and memory being initialized *once at the start*). We expand on its many intricacies in Section 5.3.2.

- `IsDstConsistent` / `IsSrcConsistent` – Capturing the axiomatic semantics of the target and source languages, respectively.
- `MappingRestricted` – Although we do not model the program syntax, we must restrict the target execution to those produced from mapped programs; for instance, if a mapping does not produce fences, there is no way for its corresponding event to appear in the target execution.
- `SyntaxMapped` – Similarly, we must capture the instruction mapping over the execution graphs. Intuitively, we "lift" the mapping from syntax to semantics. For instance, it captures every x86 R mapping to an Arm R_{Q} event (*i.e.,* acquirePC-load).
- `behavior` – Capturing the terminal state of program memory, which must be unaffected by the transformation.

5.3 WEAK MEMORY MECHANIZATION

When writing pen-and-paper proofs, many subtle details can be ignored, often to prioritize interesting parts. In contrast, proof mechanization demands meticulous specification of all details. In this section, we identify and clarify issues in mechanizing weak memory models.

These challenges are independent from transformation proofs (Section 5.4) and appear regardless of the chosen proof assistant. We thus present them generally, but give our Agda solution as reference. We use mathematical definitions for unary predicates and binary relations, which we mechanized in our separate Agda proof library Dodo.

5.3.1 *Implicit Definitions*

First, we inspect definitions that existing works (*e.g.,* by Alglave et al. (2014); Podkopaev et al. (2019)) give only *implicitly* in writing.

Locations and Values are Natural Numbers. The R and W events access memory locations and the values contained within them. In computer hardware, an (32-bit) x86 CPU can address at most 2^{32} bytes of memory, requiring memory addresses to be 32 bits in size. Additionally, the values are individual bytes (*i.e.,* 8 bits) that are individually addressable. That would imply *locations* are 32-bit integers while *values* are 8-bit integers for x86 CPU's.

Following the same line of reasoning, x86-64 and AArch64 CPUs would have 64-bit locations –or 48-bits (Intel Corporation, 2025; AMD, 2024)– with 8-bit values. However, when formalizing weak memory semantics, such details are largely irrelevant. The proofs consider the *structure* of the execution graph, rather than the terminal memory state (*i.e.,* we prove graph robustness rather than state robustness, as defined by Lahav and Margalit (2019)). Thus, locations and values should be modeled as *natural numbers*, abstracting away from architectural bit widths.

Tag `RMW` events. When an atomic `RMW` instruction succeeds, an rmw relation relates its R and W. However, the `RMW` *fails* when an another thread writes to the same location in-between the atomic read-and-write; in that case the `RMW` does not write its value, the W event is not generated, and the rmw remains absent from the execution graph.

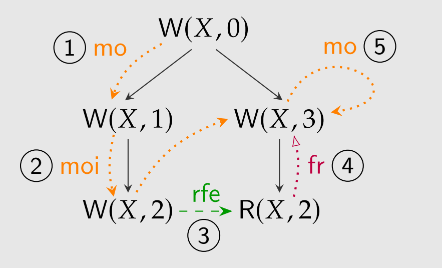
As weak memory semantics represent only the execution graph – but not the program syntax – we cannot know whether a given R event was produced by a regular read instruction or a `RMW`. However, that difference matters for mapping proofs. For instance, their instructions may map differently (*e.g.*, in Section 5.5). In that case, we must distinguish between regular and `RMW`-produced events, which we *explicitly tag* with the generating instruction's type:

```
data Tag : Set where tmov trmw : Tag
```

Initialization Events. Execution graphs often contain the initial values of memory locations at the start. The intuition is that memory has some value before the program starts. Of course, some earlier instruction has produced that value – perhaps it was another earlier program or the operating system kernel. However, when modeling program semantics, such details are irrelevant because those earlier writes cannot reorder with the program's instructions. They still *behave like* W events, because they produce values, but do not order like them. We separately denote them as W_{init}.

In contrast to regular W events, which occur in a particular thread, W_{init} events precede all other events in all threads. Additionally, they order stronger than other W events. For instance, while x86 does not order W–R pairs, it should order W_{init}–R pairs. To account for that difference, W_{init} events must explicitly be distinguished from regular W events, on any architecture.

Internal vs. External. In Section 1.2 we defined *external* relations as those between different threads, aligning with Alglave et al. (2014, 2021) and Batty et al. (2012). In contrast, *internal* relations are within a thread. Naively, we could claim the internal relations are po-related, while the external are not. To clarify what that means, consider the following execution (which is *not* well-formed):



Clearly, moi $\textcircled{2}$ is *internal*, because it follows along po in the same thread. Also, rfe $\textcircled{3}$ is clearly *external*, because it is between different threads. However, for mo $\textcircled{1}$ it is less clear, because the initialization event $\mathsf{W}(X, 0)$ is not on any thread – it precedes the threads. The source event is thus not the same thread as the target, making it *external* by the prose definition. However, the events are po-related, meaning it could be considered *internal*.

The fr $\textcircled{4}$ is ambiguous for a different reason. Clearly, it related events within the same thread, making it *internal* by the prose definition. However, it relates the events by po^{-1}, making it ambiguous whether they are therefore also po-related. Finally, mo $\textcircled{5}$ relates an event to itself (which is not well-formed), meaning it is not po-related to itself, introducing ambiguity.

Whichever choice we make does likely not affect proof validity, as mo $\textcircled{5}$ is disallowed by well-formedness, while fr $\textcircled{4}$ violates Coherence on most architectures. It does, however, affect proof internals as it determines the proof cases within which we handle them. We decide on *internal*:

$$\mathsf{int} \triangleq \mathsf{po} \cup \mathsf{po}^{-1} \cup Eq$$

where Eq is propositional equality

Conversely, external is its inverse: $\mathsf{ext}(x, y) \triangleq \neg\mathsf{int}(x, y)$. As a consequence, moi $\textcircled{1}$, fr $\textcircled{4}$, and mo $\textcircled{5}$ are all *internal*.

5.3.2 *Well-formedness*

Well-formedness formally captures an execution "making sense". It formally defines properties of an execution, many of which we list below (where E is the set of events). We discuss complicated properties separately below.

- Every event has a *unique* identifier. That is, '$x.\mathsf{uid} \equiv y.\mathsf{uid} \rightarrow x \equiv y$', for any $x, y \in \mathsf{E}$
- '$\mathsf{rf} \subseteq \mathsf{W}{\times}\mathsf{R}$', '$\mathsf{mo} \subseteq \mathsf{W}{\times}\mathsf{W}$', '$\mathsf{rmw} \subseteq \mathsf{R}^{\mathtt{trmw}}{\times}\mathsf{W}^{\mathtt{trmw}}$'
- rf, mo, and rmw are between same-location events
- rf is between same-value events
- $\mathsf{rmw} \subseteq \mathsf{po}_{\mathsf{imm}}$
- '$\mathsf{codom}(\mathsf{rmw}) \equiv (\mathsf{E}{\cap}\mathsf{W}^{\mathtt{trmw}})$' – Stating any RMW-marked W event within the execution is in the the co-domain of rmw. Note that an RMW-marked R event is *not* necessarily in codom(rmw), because the RMW instruction does not generate rmw upon failure.
- '$\mathsf{udr}(\mathsf{po}) \equiv \mathsf{E}$' – any event in the execution occurs in po, and any event in po is in the execution (*i.e.*, in E).

> *Notation.* udr is the *union of domain and range*, *i.e.*, $\mathsf{udr}(R) \triangleq \mathsf{dom}(R) \cup \mathsf{codom}(R)$.

- ‘$\mathsf{udr}(\mathsf{rf}) \subseteq \mathsf{E}$’ and ‘$\mathsf{udr}(\mathsf{mo}) \subseteq \mathsf{E}$’, stating any event in rf, respectively mo, is part of the execution (in E).
- Every R event reads from a *exactly one* W:
  - ‘$(\mathsf{R} \cap \mathsf{E}) \subseteq \mathsf{codom}(\mathsf{rf})$’ – it reads from *at least one* W.
  - ‘$\mathsf{functional}(\mathsf{rf}^{-1})$’ – it reads from *at most one* W.

    > *Notation.* $\mathsf{functional}(R) \ \triangleq\ R(x,y) \wedge R(x,z) \longrightarrow y \equiv z$

- All memory locations are initialized *exactly once*
  - ‘$\forall\, \mathsf{loc} \longrightarrow \exists x.(\ \mathsf{init}(x) \wedge x.\mathsf{loc} \equiv \mathsf{loc} \wedge x{\in}\mathsf{E}\ )$’ – *at least once*
  - ‘$\mathsf{init}(x) \wedge \mathsf{init}(y) \wedge x.\mathsf{loc} \equiv y.\mathsf{loc} \ \longrightarrow\ x \equiv y$’ for all $x, y \in \mathsf{E}$ – *at most once*

Some additional properties may be *derived* from those listed above. For instance, as fr is derived (*i.e.*, $\mathsf{fr} \triangleq \mathsf{rf}^{-1};\mathsf{mo}$), its properties can also be derived (*e.g.*, $\mathsf{fr} \subseteq \mathsf{R}{\times}\mathsf{W}$). Finally, we address several challenging properties below.

Strict Total Orders. Intuitively, po is a *strict total order per thread* and mo is a *strict total order per location*. A challenge for po is that $\mathsf{W}_{\mathsf{init}}$ events are not in threads. A challenge for both is that they are only total over events *in* E. A strict total order is transitive and *trichotomous*[21], where trichotomy states ‘$x{<}y \vee x{\equiv}y \vee x{>}y$’ (for any x and y), of which the cases are additionally *mutually exclusive*.

We separately define transitivity and trichotomy for orders. Clearly, ‘$\mathsf{transitive}(\mathsf{po})$’ and ‘$\mathsf{transitive}(\mathsf{mo})$’, but mo is only trichotomous over same-location write events (W) in E; po is only trichotomous over events in E within a given thread, or when either is an initialization event. We mechanize these properties in Agda as:

```
po-tri : ∀ (tid : ThreadId)
  → Trichotomous _≡_
    (filter-rel ((EvInit ∪₁ HasTid tid) ∩₁ events) po)
co-tri : ∀ (loc : Location)
  → Trichotomous _≡_
    (filter-rel (EvW ∩₁ HasLoc loc ∩₁ events) co)
```

Primarily, the `filter-rel` produces a subset of po satisfying the predicate. It crucially differs from intersection, which retains trichotomy over *any* event, regardless of whether it satisfies the predicate. Instead, our definition of `filter-rel` lifts the predicate to the type, ensuring trichotomy only holds for those events satisfying the predicate; we mechanized its definition in our separate proof library Dodo, as it is independent from weak memory.

[21] Observe that trichotomy implies asymmetry and irreflexivity.

Initialization First. Within both strict total orders over po and mo, initializations occur strictly *first* in the order. We formalize this with `⊤-Precedes-⊥` axiom included in Dodo, formally stating '$R(x, y) \to (P(y) \to P(x))$' (for any x, y):

```
po-init-first : ⊤-Precedes-⊥ EvInit po
co-init-first : ⊤-Precedes-⊥ EvInit co
```

Constructive Logic. Agda implements a dependent type theory (Bove et al., 2009), corresponding to a constructive (or intuitionistic) logic (Martin-Löf, 1982) – *e.g.*, like Rocq (Bertot and Castran, 2010). Consequently, it lacks the *Law of the Excluded Middle* (LEM, *i.e.*, $P \lor \neg P$). While that guarantees *computability* of proofs, it makes working with set-theoretic (*i.e.*, non-constructive) axiomatic weak memory models challenging. Instead of postulating LEM in general, we introduce it strategically for definitions that can constructively satisfy it, thus retaining computability (and proof checking with `--safe` flag[22]).

Concretely, we use it to *decide* whether an R event originates from an `RMW` instruction. After all, regular reads may order differently from successful `RMW`-R events or order by a different rule. Additionally, we use it to *decide* whether an event with unique ID exists in E or not (*i.e.*, decide whether '$\exists x \,.\, x.\mathsf{uid} \equiv \mathsf{uid} \land x \in \mathsf{E}$' for any uid).

Splittable Orders. Another property related to *strict total orders* is decidability of $\mathsf{po}_{\mathsf{imm}}$. While po is both strictly total and transitive, proofs often also need to "peel off" a po-adjacent event. Consider the example below, with arbitrary po-related events.

$$\begin{tikzcd}
x \arrow[d, "\mathsf{po}"'] & p \arrow[d] \\
y \arrow[d] & q \\
z &
\end{tikzcd}$$

Clearly, po is decidable for x and z, as we can establish $\mathsf{po}(x, z)$ by totality of po. However, that does not mean $\neg\mathsf{po}_{\mathsf{imm}}(x, z)$ is decidable, as it requires knowledge of intermediate event y; or, for p and q, where $\mathsf{po}(p, q)$, knowing $\mathsf{po}_{\mathsf{imm}}(p, q)$ demands knowing no such intermediate events exists. Unfortunately, we cannot *assume* its existence or nonexistence in a constructive logic (as it requires LEM). Instead, we can *define* it:

$$\mathsf{po} \;\equiv\; (\mathsf{po}_{\mathsf{imm}})^{+} \qquad \text{(SplittablePo)}$$

It states that for any po edge, we can always decompose it into a chain of $\mathsf{po}_{\mathsf{imm}}$ edges. For any instantiation of a finite execution, we can *constructively* provide SplittablePo. This definition allows us to "peel off" individual $\mathsf{po}_{\mathsf{imm}}$ edges in our proofs. For instance, given $\mathsf{po}(x, z)$, we can now obtain y satisfying $\mathsf{po}(x, y) \land \mathsf{po}_{\mathsf{imm}}(y, z)$. We also

[22] The `--safe` flag prohibits postulations (among other things), meaning a reader can trust that no postulations occur throughout the proof.

mechanized this generalized definition and related properties in Dodo – *i.e.*, generally, as SplittableOrder, where $\mathsf{SplittablePo} \triangleq \mathsf{SplittableOrder}(\mathsf{po})$. One such property states SplittableOrder is *strictly stronger* than transitivity, which is required for po's strict total order.

5.3.3 *Architecture Definition: Armv8*

While the well-formedness definitions from the previous subsection are shared between models, various architectures and languages additionally include their own events, primitive relations, and consistency axioms. Here we elaborate on our process of mechanizing the Armv8 model[23] by Alglave et al. (2021); Alglave and Maranget (2025). However, we do not include the full Armv8 model here because of its large size, but instead highlight some of its mechanization challenges below.

While all architectures have R, W, and F events, architectures provide different variants. Of course, all *read* instructions read and *write* instructions write, but their variants *order differently*. On Arm, W is a *unordered* write; W_L is a *store-release*, which orders with preceding events; R is a *unordered* read; R_A is a *load-acquire*, which orders with subsequent events *and* preceding W_L events; and R_Q is a *load-acquirePC*, which also orders with subsequent events, but *not* with preceding W_L.

In our Agda mechanization we similarly annotate the events with those *labels*. The Armv8 R/W labels are:

```
data LabR : Set where
  lab-r : Tag → LabR -- ^ regular read
  lab-a : Tag → LabR -- ^ acquire read
  lab-q : LabR       -- ^ acquirePC read

data LabW : Set where
  lab-w : Tag → LabW -- ^ regular write
  lab-l : Tag → LabW -- ^ release write
```

The labels *explicitly* store their `Tag` (from Section 5.3.1). We cannot store those tags externally from the labels, as some do not have them; for instance, an R_Q (*i.e.*, `lab-q`) *cannot* be generated from a `RMW` and is thus always `tmov`.

Of course, the events themselves do not enforce the ordering, which is enforced by the *axioms*. Similarly to x86's XHB constraint, Arm orders *some* po edges. We mechanize those constraints in Agda as follows (abbreviated):

[23] Here, we use a differently refactored variant of the model by Alglave et al. (2021) presented in Chapter 1, taken from the author's GitHub repository (Alglave and Maranget, 2025).

```
-- | Barrier Ordered Before
data Bob (x y : Event) : Set where
  bob-acq : ( ⦗ EvA ∪₁ EvQ ⦘ ⨾ po )  x y → Bob x y
  bob-rel : ( po ⨾ ⦗ EvL ⦘ )         x y → Bob x y
  bob-la  : ( ⦗ EvL ⦘ ⨾ po ⨾ ⦗ EvA ⦘ ) x y → Bob x y
  -- ...

-- | Locally Ordered Before (immediate)
data Lobᵢ (x y : Event) : Set where
  lobi-bob : Bob x y → Lobᵢ x y
  -- ...

-- | Ordered Before (immediate)
data Obᵢ (x y : Event) : Set where
  obi-lob : (TransClosure Obᵢ) x y → Obᵢ x y
  -- ...

record IsArmv8Consistent : Set where
  field
    ax-external : Irreflexive _≡_ (TransClosure Obᵢ)
    -- ...
```

Observe we use predicates for types of events – *e.g.,* `EvA` – which are inhabited for events with the corresponding label, *e.g.,* `lab-a`. The architecture consistency rules are included in `IsArmv8Consistent`, within which we only show the "*external visibility*" constraint (Alglave et al., 2021; Alglave and Maranget, 2025) in `ax-external`.

5.4 MAPPING PROOF MECHANIZATION

In the previous section, we addressed general mechanization challenges for axiomatic weak memory, which we solve in our proof framework Burrow. However, Burrow solves a bigger problem: It provides domain-specific abstractions and primitives for weak memory transformation proofs!

With those abstractions we can prove fragments *generally*. In Section 5.2.1 we saw that proving a transformation correct requires at least *(1)* constructing a source execution, *(2)* proving well-formedness, *(3)* proving consistency, and *(4)* proving behavior preservation. While constructing an execution and proving consistency is specific to a mapping, the others are largely similar. In particular, well-formedness consists of 27 properties, which is time-consuming to prove for every new mapping. However, if we select the right abstractions, we can prove properties generally.

5.4.1 *Mapping Primitives*

When mapping between two executions, we often translate properties. For instance, if x^{t} in the target is a R event, then after mapping it to x^{s} in the source, x^{s} is *also* a R event; the property remains preserved. Similarly, those properties must also be preserved in the other direction, from source to target. Proving that properties and relations remain preserved along the mapping is a significant part of proving well-formedness and consistency. In the mechanization, we thus expect definitions resembling the following:

```
ev[⇐] : DstEvent → SrcEvent
ev[⇒] : SrcEvent → DstEvent

R[⇐] : {xᵗ : DstEvent} → EvR xᵗ → EvR (ev[⇐] xᵗ)
R[⇒] : {xˢ : SrcEvent} → EvR xˢ → EvR (ev[⇒] xˢ)
```

With such definitions we can translate properties between the source and target execution. However, `ev[⇐]` and `ev[⇒]` must be *inverses* of each other, meaning they *define* each other. In other words, after *defining* one, we can *derive* the other. As we are given a target execution, from which we produce the source, users need only define `ev[⇐]`.

Another problem remains: The mapped events are not necessarily in the execution (*i.e.*, in E). If we produce our source execution with the `ev[⇐]` definition above, arbitrary events could appear. Instead, we need to *restrict* the `ev[⇐]`-mapped events to those actually contained in the target:

```
ev[⇐] : {xᵗ : DstEvent} → xᵗ ∈ events dst → SrcEvent

R[⇐] : {xᵗ : DstEvent}
  → (x∈dst : xᵗ ∈ events dst) → EvR xᵗ
  → EvR (ev[⇐] x∈dst)
```

This allows users to define `ev[⇐]`, *only over events in the target execution*, from which we derive `ev[⇒]` generically.

Deriving Properties. `R[⇐]` proves `ev[⇐]` preserves that the mapped event is a R. While we can derive `ev[⇒]` from `ev[⇐]`, we *cannot* derive `R[⇒]` from `R[⇐]`; because it is *not generally true*; for instance, consider our read-after-write elimination from Section 2.6.2, where the target has one R event less. In contrast, for mappings between architectures, R-events remain R-events (*e.g.*, in Section 5.5). As the preservation of R along `ev[⇒]` varies between mappings, we must prove it separately for each.

A formalization issue appears when we define `R[⇒]`: "*green slime*"[24] (McBride, 2012), where a function argument is not in constructor form; *e.g.*, given $\mathtt{f}(x)$ as argument, where '$\mathtt{f} : \mathbb{N} \longrightarrow \mathrm{Bool}$', Agda does not know which of `true` and `false` are possible without knowing x. In our case, that issue appears when mirroring `R[⇐]`:

[24] *green slime* is a metaphor, commonly used by Agda developers.

```
R[⇒] : {xˢ : SrcEvent}
  → (x∈src : xˢ ∈ events src) → EvR xˢ
  → EvR (ev[⇒] x∈src)
```

The problem is that `x∈src` – which is a proof for `xˢ` being in the source execution – is also *derived* along `ev[⇐]` and is thus based on some unknown event *in the target*. Implementing `R[⇒]` becomes cumbersome, as we must pattern-match on all target events that *could potentially* produce our event `xˢ` along the function `ev[⇐]`. Instead, we implement a variant of `R[⇒]` that is defined *over the target* event:

```
R[$⇒] : {xᵗ : DstEvent}
  → (x∈dst : xᵗ ∈ events dst) → EvR (ev[⇐] x∈dst)
  → EvR xᵗ
```

That definition is much easier to implement, as it avoids the above issues. Surprisingly, we can obtain `R[⇒]` from it, using our definition of well-formedness.

Generalization. As we map many properties, whose mapping types follow a similar structure, we can define mapping *types* generally:

```
Pred[⇐] : REL₀ Predˢ Predᵗ
Pred[⇐] Pˢ Pᵗ = ∀ {xᵗ}
  → (x∈dst : xᵗ ∈ events dst)
  → Pᵗ xᵗ
  → Pˢ (ev[⇐] {xᵗ} x∈dst)
```

```
Pred[$⇒] : REL₀ Predˢ Predᵗ
Pred[$⇒] Pˢ Pᵗ = ∀ {xᵗ}
  → (x∈dst : xᵗ ∈ events dst)
  → Pˢ (ev[⇐] {xᵗ} x∈dst)
  → Pᵗ xᵗ
```

Observe that applying those definitions gives the types of `R[⇐]` and `R[$⇒]` from before:

```
R[$⇒] : Pred[$⇒] EvR EvR
R[⇐] : Pred[⇐] EvR EvR
```

We use this same idea to map relations. For instance, if mo(x^s, y^s) holds in the source, mo(x^t, y^t) should hold in the target, after mapping x^s to x^t and y^s to y^t. Our proofs also include types for *relations*, *e.g.*, satisfying '`co[⇐] : Rel[⇐] (co src) (co dst)`'. The advantage of these `[⇐]`/`[$⇒]` variants is that the programmer *implements* them *over target events*. Their implementation pattern matches over the same constructors as the definition of `ev[⇐]`. Fortunately, the programmer can *use* the `[⇒]` variant in their proofs, which is defined *over source events*. Concretely, Burrow can translate between these variants – implemented using our strategically picked decidable properties (from Section 5.3.2) – thus avoiding the "*green slime*" from above.

5.4.2 *Deriving Definitions*

When writing a mapping proof, a Burrow user first defines the mapping of events with `ev[⇐]`, according to their intended transformation. Subsequently, for architecture mapping proofs (which do not remove events), users prove `ev[⇐]` preserves the following properties back-and-forth: uid, tid, loc, val, $\mathsf{W}_{\mathrm{init}}$, tagged W, and tagged R.

In contrast, F is not needed, because it does not appear in well-formedness. From those definitions, Burrow can derive large parts of the mapping proofs *generally*:

- *The source execution* $\langle X^s$, mo^s, rf^s, rmw$^s\rangle$ – Note that po, mo, rf, rmw can be fully preserved along `ev[⇐]`, provided that the relevant event properties are. For instance, if W, loc, and val are preserved, we can also preserve mo. Other details need not be preserved (*e.g.,* the difference between regular- and release-stores on Arm).

- *Well-formedness of the source* – For the produced source execution, we can prove its well-formedness *generally*, covering all its 27 properties.

- *Behavior* – We can generally prove that the behavior, which is the terminal memory state, is preserved. This is because the user preserved Ws with their loc and val, while our constructed source execution preserves mo.

Removing Boilerplate. We can thus produce *three out of four* fragments of the proof generally. We cannot, however, prove *consistency* generally, as it is specific to the particular transformation and often the most interesting part of the proof. After all, an important design criterion of Burrow is avoiding proof "boilerplate code" for its *un*-interesting proof fragments.

Optimizations. As architecture mapping proofs *do not* remove instructions – and thus events – well-formedness can be proven over the derived mapping of po, mo, rf, and rmw. However, for optimizations that *remove* instructions – and thus events – we cannot generally derive the mapped relations. For instance, consider the read-after-write elimination from Section 5.1, which discards a R event in the target. Thus, producing the source execution and establishing its well-formedness is more challenging than before.

Because the read-after-write elimination discards a R event, its proof must remove its corresponding rf edge from any cycle. In contrast, as all W events remain, mo remains too. Additionally, the discarded R events is a non-`RMW` event, meaning all rmw edges also remain. So, although we cannot produce the source execution and its well-formedness as easily as before, still much of the proof internals remain usable (*i.e.,* for po, mo, rmw, and most well-formedness properties).

This is similarly true for other elimination proofs, which preserve *most* of the source structure; *e.g.,* a write-after-write elimination discards a W event while preserving all R events. To simplify the proofs, Burrow exposes several such definitions *generally*, whereas the programmer can adapt others to their specific elimination proof.

Skip Events. An elimination proof *removes* an event x from the graph, which is thus absent from the target execution. While the source execution is *constructed from* that target execution, there exists no event in the target producing x in the source. For instance, consider producing the eliminated R in the source in read-after-write elimination (in Section 5.1). The eliminated event x must thus be produced in the source by *another mechanism* than ev[⇐]. Similarly, source relations must be defined such that they contain x, which is particularly cumbersome for the *orders* (*i.e.*, po and mo), where transitivity and totality / trichotomy must explicitly consider x.

To avoid all that complexity, we use *skip* events, which are distinct from R, W (and W_{init}), and F events. Intuitively, they behave as-if generated by a "no-op" instruction. They appear in the execution but do not read, write, nor enforce any ordering. Consequently, they appear only in the set of events (*i.e.*, E) and in po. However, skip events significantly simplify elimination proofs when included in the target execution. The eliminated source event x then maps to a skip event. All (rf / mo / rmw) relations must still divert from x, but orders passing through them can often ignore them.

Consistency. Although Burrow cannot prove consistency generally, it can make it easier to prove. It does so by providing mapping primitives of type Pred[⇒] and Rel[⇒], which are convenient when mapping a cycle from the source to the target execution, which is the primary challenge when proving consistency. Additionally, Burrow contains functions to *lift* mappings over common structures. For instance, ⁺[⇒]¹ lifts a mapping of a relation R to a mapping of its transitive closure R^+, while ∩₂[⇒] lifts a mapping of two relations R and Q to a mapping of their intersection $R \cap Q$. In our case study we use Burrow to prove consistency for a mapping.

5.5 CASE STUDY: MAPPING X86 TO ARM

To evaluate the effectiveness of Dodo and Burrow, we prove an existing conjectured mapping from x86 to Armv8 (Alglave et al., 2021, Section 2.5). Crucially, the mapping was originally incorrect and subsequently fixed by *changing the Armv8 model* (Alglave, 2022) (as we explained in Section 3.3.3). To the best of our knowledge, our formal result of proving that mapping correct on the fixed Arm model is novel. The instruction-level mapping is given as follows:

x86		Armv8
RMOV	→	LDAPR
WMOV	→	STLR
MFENCE	→	DMBFF
CMPXCHG	→	$\mathtt{CAS}_{\mathtt{AL}}$

We previously explained those primitives in Section 1.3.3. We can only prove the mapping correct with the weak memory semantics of both x86 and Armv8. The x86-TSO model (Owens et al., 2009), which we already gave in Section 1.3.2, is simple. It consists of the Coherence, Atomicity, and XHB axioms. The Arm model (Alglave et al., 2021; Alglave and Maranget, 2025), however, is more complex. In particular, it is structured differently from the x86 model and *refactored* definitions from the previous Armv8 model (Alglave et al., 2021) we explained in Section 1.3.3; however, we use the new model because it includes semantics for the new $\mathtt{CAS}_{\mathtt{AL}}$ instruction, which

$$
\begin{array}{ll}
\mathsf{ob} \text{ is irreflexive} & (\mathsf{external}) \\
\quad \text{where } \mathsf{ob} \triangleq (\mathsf{obs} \cup \mathsf{lob} \cup [\mathsf{R}];\mathsf{po}_{\mathsf{loc}};\mathsf{fre})^{+} & \\
\qquad\qquad \mathsf{obs} \triangleq \mathsf{rfe} \cup \mathsf{fre} \cup \mathsf{moe} & \\
\qquad\qquad \mathsf{lob} \triangleq (\mathsf{bob} \cup \mathsf{po}_{\mathsf{loc}};[\mathsf{W}] \cup \ldots)^{+} & \\
\qquad\qquad \mathsf{bob} \triangleq \mathsf{po};[\mathsf{F}_{\mathsf{SC}}];\mathsf{po} \cup [\mathsf{R}_{\mathsf{A}} \cup \mathsf{R}_{\mathsf{Q}}];\mathsf{po} \cup \mathsf{po};[\mathsf{W}_{\mathsf{L}}] & \\
\qquad\qquad\qquad \cup\, [\mathsf{W}_{\mathsf{L}}];\mathsf{po};[\mathsf{R}_{\mathsf{A}}] \cup [\mathsf{codom}([\mathsf{R}_{\mathsf{A}}];\mathsf{amo};[\mathsf{W}_{\mathsf{L}}])];\mathsf{po} & \\
\qquad\qquad\qquad \cup \ldots & \\
([\mathsf{R}];\mathsf{po}_{\mathsf{loc}};[\mathsf{W}];\mathsf{rfi};[\mathsf{R}]) \text{ is irreflexive} & (\mathsf{internal\text{-}rw}) \\
([\mathsf{W}];\mathsf{po}_{\mathsf{loc}};[\mathsf{W}];\mathsf{moi};[\mathsf{W}]) \text{ is irreflexive} & (\mathsf{internal\text{-}ww}) \\
([\mathsf{W}];\mathsf{po}_{\mathsf{loc}};[\mathsf{R}];\mathsf{fri};[\mathsf{W}]) \text{ is irreflexive} & (\mathsf{internal\text{-}wr})
\end{array}
$$

Figure 25: Arm model (Alglave et al., 2021; Alglave and Maranget, 2025), abbreviated. Arm additionally satisfies Atomicity (from Section 1.3).

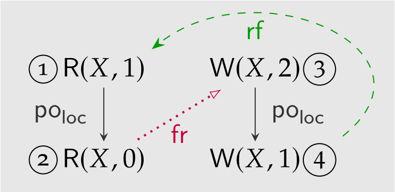
generates an $[\mathsf{R}_{\mathsf{A}}];\mathsf{amo};[\mathsf{W}_{\mathsf{L}}]$ relation, where $\mathsf{amo} \subseteq \mathsf{rmw}$. We give the subset of the Arm model that is relevant for our proofs in Figure 25.

The Arm model factors internal and external relations differently from x86. In particular, x86's Coherence constraint includes both internal and external rf, mo, fr events. Those include rfi edges, which are an W–R pair that is not otherwise ordered in x86 (*i.e.*, not in xppo). In contrast, Arm's internal-rw/ww/wr constraints *exclusively* contain internal relations. Similarly, unlike x86's Coherence, Arm's external constraint is strictly external as it does not include rfi, fri, or moi edges. Those mismatches between the model structures makes writing the mapping proofs challenging.

Fortunately, Burrow simplifies constructing the source execution. Through Burrow's domain-specific primitives and abstractions, it proves well-formedness and behavior preservation as instances of general proofs. Only the proofs for the consistency axioms remain. We follow the typical proof structure from Section 5.2.1, proving consistency of the source execution, given that the target execution is consistent. Thus, we demonstrate that any violation of an x86 axiom violates an Arm axiom.

As both architectures satisfy Atomicity, its proof is a trivial mapping. What remains is x86's Coherence and XHB, of which we give a proof outline to illustrate its complexities. Naturally, our artifact includes the full mechanization in Agda, on whose implementation we reflect below.

Coherence. For x86's Coherence axiom, we are given a '$\mathsf{po}_{\mathsf{loc}} \cup \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{mo}$' cycle. To follow along with the proof, consider the example cycle in x86 below, starting at ①. We omit events and edges of the graph that are outside the cycle.



The proof sketch steps are then as follows:

1. Rotate the chain to start at a W event. Note that such a W event always exists. Every rf, fr, and mo edge has at least one W event. If the entire cycle were only $\mathsf{po_{loc}}$, it is not well-formed because po is irreflexive.

   > *Example:* $\textcircled{3} \xrightarrow{\mathsf{po_{loc}}} \textcircled{4} \xrightarrow{\mathsf{rf}} \textcircled{1} \xrightarrow{\mathsf{po_{loc}}} \textcircled{2} \xrightarrow{\mathsf{fr}} \textcircled{3}$

2. Convert it to '$\mathsf{po_{loc}} \cup \mathsf{rfe} \cup \mathsf{fre} \cup \mathsf{moe}$'. We can split off the rfi, fri, and moi edges. If they are po-forward, we include them in $\mathsf{po_{loc}}$. If they are po-backward (*i.e.,* po^{-1}), we have internal cycles, which trivially map to Arm and are disallowed by internal-rw/ww/wr.

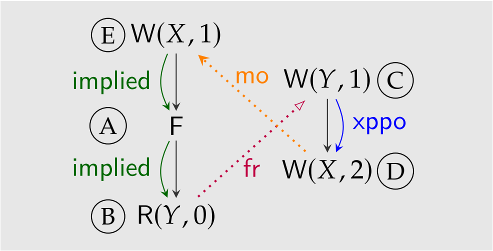
   > *Example:* $\textcircled{3} \xrightarrow{\mathsf{po_{loc}}} \textcircled{4} \xrightarrow{\mathsf{rfe}} \textcircled{1} \xrightarrow{\mathsf{po_{loc}}} \textcircled{2} \xrightarrow{\mathsf{fre}} \textcircled{3}$

3. We now have a '$[\mathsf{R}{\cup}\mathsf{W}];(\mathsf{po_{loc}}{\cup}\mathsf{rfe}{\cup}\mathsf{fre}{\cup}\mathsf{moe})^{+};[\mathsf{W}]$' chain, from which we produce '$(\mathsf{po_{loc}};[\mathsf{W}]) \cup ([\mathsf{R}];\mathsf{po_{loc}};\mathsf{fre}) \cup \mathsf{moe} \cup \mathsf{rfe} \cup \mathsf{fre}$'. In particular, we require the terminal W (from step 1) when $\mathsf{po_{loc}}$ is the final edge, producing '$\mathsf{po_{loc}};[\mathsf{W}]$'.

   > *Example:* $\textcircled{3} \xrightarrow{\mathsf{po_{loc}};[\mathsf{W}]} \textcircled{4} \xrightarrow{\mathsf{rfe}} \textcircled{1} \xrightarrow{[\mathsf{R}];\mathsf{po_{loc}};\mathsf{fre}} \textcircled{3}$

4. Now it maps to Arm's ob and is forbidden by external.

XHB. For x86's XHB axiom, we are given a '$\mathsf{xppo} \cup \mathsf{implied} \cup \mathsf{rfe} \cup \mathsf{fr} \cup \mathsf{mo}$' cycle. To follow along with the proof, consider the following cycle in x86, starting at Ⓐ.



The proof sketch follows the following steps:

1. Rotate it to start at a R or W event. Only the implied relation may end in a F, which is necessarily followed by another implied. Hence, a R or W eventually appears, or is entirely in po which well-formedness disallows.

   $$\textcircled{B} \xrightarrow{\mathsf{fr}} \textcircled{C} \xrightarrow{\mathsf{xppo}} \textcircled{D} \xrightarrow{\mathsf{mo}} \textcircled{E} \xrightarrow{\mathsf{implied}} \textcircled{A} \xrightarrow{\mathsf{implied}} \textcircled{B}$$

2. Convert it to ‘$\mathsf{xppo} \cup [\mathsf{R} \cup \mathsf{W}];\mathsf{implied}^+;[\mathsf{R} \cup \mathsf{W}] \cup \mathsf{rfe} \cup \mathsf{fre} \cup \mathsf{moe}$’. The $\mathsf{R} \cup \mathsf{W}$ bounds on implied necessarily exist, for the same reason as in step 1.

   $$\textcircled{B} \xrightarrow{\mathsf{fre}} \textcircled{C} \xrightarrow{\mathsf{xppo}} \textcircled{D} \xrightarrow{\mathsf{moe}} \textcircled{E} \xrightarrow{[\mathsf{R} \cup \mathsf{W}];\mathsf{implied}^+;[\mathsf{R} \cup \mathsf{W}]} \textcircled{B}$$

3. Now it maps to Arm’s ob and is forbidden by external. The $\mathsf{R} \cup \mathsf{W}$-bounded $\mathsf{implied}^+$ maps to bob^+, as it either has an intermediate F or a $\mathtt{CAS}_{\mathtt{AL}}$-produced event.

   $$\textcircled{B} \xrightarrow{\mathsf{obs}} \textcircled{C} \xrightarrow{\mathsf{bob}} \textcircled{D} \xrightarrow{\mathsf{obs}} \textcircled{E} \xrightarrow{\mathsf{bob}} \textcircled{B}$$

Mechanization Challenges. Although the above proof sketch is short, the entire consistency proof is much larger (over 700 source lines). The remaining mechanization complexity resides in matching on many nested cases of the x86 relations. In particular, step 3 of the Coherence proof requires traversing arbitrary $\mathsf{po}_{\mathsf{loc}}$ chains, followed by any of the other cases, each interaction requiring attention. Similarly, step 2 in the XHB proof requires traversing arbitrary implied chains, followed by any other case. Finally, step 3 of the XHB proof is conceptually simple, but covers many cases. Additionally, the chain may traverse through a $\mathsf{W}_{\mathsf{init}}$, which is not well-formed because no implied, rfe, fre, or moe can target it. Only xppo can target it from another $\mathsf{W}_{\mathsf{init}}$ event, merely shifting the problem to that event, eventually reaching one of the other cases. Demonstrating that no such $\mathsf{W}_{\mathsf{init}}$ event exists requires inspecting all cases.

Q.E.D. Burrow simplified the proof mechanization efforts greatly. In particular, Burrow reduces the “boilerplate” code needed to prove well-formedness and behavior preservation, by *proving it generally*. For this mapping we only specifically *defined the mapping* and *proved consistency* – while relying on Burrow for the remaining parts – formally proving the mapping under the fixed Arm model. □

5.6 RELATED WORK

Semantics & Proofs. Much work on formal weak memory semantics precedes ours. For instance, Alglave et al. (2014) propose axiomatic semantics for Arm, which their Herd tool extensively tests on Arm hardware. While they do not write *mapping* proofs, they prove the equivalence of their axiomatic and operational models in Rocq.

Later work proposes formal weak memory semantics for C++ (Batty et al., 2011), mechanized in Isabelle/HOL (Nipkow et al., 2002), but parts of their proofs are only on pen-and-paper. Some of their results were later found incorrect and fixed by Lahav et al. (2017), along with more pen-and-paper proofs.

The *intermediate memory model* (IMM) by Podkopaev et al. (2019) has mechanized proofs. IMM reduces the proof burden as an *intermediate* model; given N source and M target languages, only $N+M$ mapping proofs are needed instead of $N \times M$. Unlike our Burrow, IMM does not address the mechanization challenges inherent in *writing* those $(N+M)$ proofs. Separately, IMM uses their Rocq library *Hahn* (Vafeiadis, 2018) with lemmas and tactics for binary relations, mirroring our library Dodo.

Tools. ArMOR by Lustig et al. (2015) detects ordering violations on Arm at runtime, but does not provide formal guarantees and cannot capture all semantics. Chakraborty and Vafeiadis (2016) validate that LLVM optimizations for C11 programs do not incorrectly reorder weak memory primitives. GenMC by Kokologiannakis and Vafeiadis (2021) is a model checker that detects reordering violations by exhaustively exploring a program's state space under weak memory. Similarly, Dartagnan by Gavrilenko et al. (2019) is a bounded model checker that checks assertions under weak memory with an SMT solver. Recently, the FDR4 model checker by Gibson-Robinson et al. (2014) has been used for bounded refinement checking (Raad et al., 2024). C11Tester by Luo and Demsky (2021) explores many executions to detect violations under weak memory. Kokologiannakis et al. (2023) propose Kater to *automatically* decide whether a mapping is correct, among other things. Model definitions must often be reformulated beforehand to support automation; for instance, to allow rotating a sequence of events or to map one source axiom to multiple target axioms (as we did in Section 5.5). In contrast, mechanized proofs ensure even those initial reformulations are correct, as part of the full mapping proof.

5.7 CONCLUSION

We propose Burrow, a domain-specific Agda proof framework for weak memory transformation proofs. We identified and clarified several challenges in mechanizing axiomatic weak memory models and their corresponding transformation proofs, which we addressed in Burrow. Additionally, we developed Dodo, which is an Agda proof library for predicates and binary relations, independent from weak memory proofs but internally used by Burrow. Finally, Burrow contains domain-specific abstractions and primitives for weak memory proofs, with which it proves large fragments of weak memory proofs *generally*. We demonstrate its effectiveness by proving a conjectured mapping from x86 to Arm. The proved mapping differs from our earlier mappings (Chapters 2 to 4) because it relies upon specific load and store primitives, whereas our mappings inserted fences.

While Burrow's development succeeds our proofs for Lasagne (Chapter 2) and Risotto (Chapter 3), we believe it would have significantly reduced their required proof engineering effort. We did use an early version of Burrow for the development of the Arancini (Chapter 4) proofs, significantly reducing their implementation effort. We expect Burrow will similarly simplify the mechanization of other weak memory transformation proofs.

Chapter 6

Porting Programs with Dynamic Analysis

ABSTRACT In this chapter, we propose a dynamic analysis technique to automatically check if the program would exhibit any additional execution on the new weaker architecture Arm, compared to x86. Upon finding such an execution, we transform the program to forbid such executions. We implemented our algorithm in our ORIGAMI system. We evaluate on several real-life concurrency benchmarks to demonstrate the effectiveness of our approach. ORIGAMI identifies cases where new executions are possible on Arm compared to x86 while being more precise than state-of-the-art approaches, all while not reporting spurious errors. Our approach generates more efficient programs that show, on average, 1.8× performance improvement (up to 8×) compared to existing approaches.

6.1 INTRODUCTION

To forbid unanticipated weak memory behaviors in Arm, programs must be transformed by strengthening memory accesses or inserting fences. These transformations must be performed judiciously, as the reinforced memory accesses and fences are costly to performance (Liu et al., 2020). In Chapters 2 to 4 we defined mapping schemes that are correct *in general*, for any program. However, for a specific program they often insert too many fences, meaning we could do better by using what we know about that program. In particular, when porting a concurrent program from *strong-to-weak* concurrency models, we could analyze *which* additional executions the program demonstrates on Arm, and then transform the program to avoid only those observed executions.

Exploring many executions following a weak memory model poses a key challenge to analysis techniques. Existing approaches apply various analysis and verification techniques (Chakraborty, 2021; Oberhauser et al., 2021; Beck et al., 2023; Bouajjani et al., 2013a; Abdulla et al., 2015a,b; Shasha and Snir, 1988; Lee and Padua, 2001; Linden and Wolper, 2011, 2013; Alglave et al., 2017) which face tradeoffs between precision and scalability. Static analysis-based approaches (Chakraborty, 2021; Oberhauser et al., 2021; Shasha and Snir, 1988; Lee and Padua, 2001; Linden and Wolper, 2011, 2013; Alglave et al., 2017) may over-approximate possible executions and, in turn, insert additional fences that result in suboptimal solutions. Some approaches (Oberhauser et al., 2021; Bouajjani et al., 2013a; Abdulla et al., 2015a,b) use model checkers to explore executions. However, model checkers do not scale to large concurrent applications. Thus, *any approach* will inherently miss violations, for instance, because they emerge beyond the reachable scale. However, we need a pragmatic approach that scales to larger programs, even when it cannot guarantee completeness.

Contributions. This chapter addresses the problem of porting concurrent programs written for x86 to Arm. We model the problem as robustness analysis and enforcement (Chakraborty, 2021) and use *dynamic analysis* to obtain a solution that scales to larger programs.

- **Operational-Axiomatic Semantics for x86 and Arm** – We define an operational-axiomatic weak memory semantics for x86 and Arm (Section 6.3), based on existing axiomatic semantics by Owens et al. (2009) and Alglave et al. (2014, 2021).
- **Dynamic Analysis Algorithm** – Following those operational-axiomatic semantics, we develop our dynamic analysis algorithm to detect x86-Arm robustness violations (Section 6.4). To do so, it constructs an execution under x86's consistency model and checks if it has any potential step that violates the stronger x86 model while remaining consistent under Arm's weaker model.
- **Program Repair Algorithm** – Second, we propose an algorithm that locates the program operations that resulted in the x86-Arm mismatch, allowing us to minimally strengthen these operation to forbid the additional executions (Section 6.5).
- **Origami Implementation** – Third, we propose Origami, which implements both these algorithms to dynamically detect violations and propose fixes on real programs.
- **Experimental Evaluation** – Finally, we evaluate Origami on several concurrency benchmarks and compare with state-of-the-art approaches Fency by Chakraborty (2021) and our Lasagne and Risotto (Chapters 2 and 3) mapping schemes (Section 6.6). Our analysis identifies several programs that exhibit more executions in Arm than in x86, which we subsequently fix.

Thus Origami is *(1)* more precise in several cases compared to existing approaches, *(2)* scales to larger programs, *(3)* demands fewer concurrency primitives while transforming the program, and therefore *(4)* produces programs that are significantly faster than those generated by other approaches.

6.2 BACKGROUND

We use the same syntax and weak memory semantics for x86 and Arm programs as introduced in Chapter 1. Here we highlight some challenges specific to our approach.

6.2.1 *Challenges in Porting Concurrency*

Consider the program in Figure 26, which is extracted from the concurrent `mpmc-queue` benchmark, where `X` and `Y` are shared locations initialized to zero. Suppose the program was written for x86. While the high-level source language (*i.e.*, `mpmc-queue` implemented in C++) may include weakly-annotated primitives, they are compiled to their only corresponding variant available on x86 and Arm. For instance, a C++

$$\begin{array}{l||l} a = \mathtt{RMW}(\mathtt{X},0,1); & c = \mathtt{RMW}(\mathtt{Y},1,0); \\ b = \mathtt{RMW}(\mathtt{Y},0,1); & d = \mathtt{X}; \end{array}$$

(a) x86

$$\begin{array}{l||l} a = \mathtt{RMW}_{\mathtt{AL}}(\mathtt{X},0,1); & c = \mathtt{RMW}_{\mathtt{rlx}}(\mathtt{Y},1,0); \\ b = \mathtt{RMW}_{\mathtt{rlx}}(\mathtt{Y},0,1); & d = \mathtt{X}_{\mathtt{Q}}; \end{array}$$

(b) Arm

Figure 26: x86 and Arm versions of the program extracted from `mpmc-queue`. The outcome $c = 1, d = 0$ is allowed in Arm but not in x86. Strengthening the access from $\mathtt{RMW}_{\mathtt{rlx}}$ to $\mathtt{RMW}_{\mathtt{A}}$ in the second thread will forbid the outcome in Arm as well. The resulting program has the same concurrent behaviors in x86 and Arm.

$\mathtt{RMW}_{\mathtt{rlx}}$ compiles to a $\mathtt{RMW}_{\mathtt{rlx}}$ on Arm but to a stronger $\mathtt{RMW}$ on x86. So, when the same source program is compiled to Arm, it suddenly executes more instructions out-of-order – in this case, the two accesses in the second thread.

In particular, the accesses in the first thread are ordered on x86 and Arm, but those in the second thread are only ordered on x86 but *not* on Arm. Consequently, the additional outcome $c = 1$ and $d = 0$ appears when recompiling to Arm, which was impossible before on x86. Hence, naively recompiling an existing source program, which was only run *or tested* on x86 may show additional unforeseen behavior after recompiling to Arm.

Challenges and our approach. Porting programs from a stronger architecture (*e.g.*, x86) to a weaker one (*e.g.*, Arm) presents two main challenges. The first challenge is identifying whether the ported program has more behaviors than the original (as in the example above). These additional behaviors must be eliminated, as they violate the programmer's (intended) invariants and threaten program correctness. This is achieved by inserting fences and/or strengthening certain memory accesses. However, such transformations slow down program execution. Thus, the second challenge is applying such transformations with minimal impact on performance.

In our example in Figure 26b, the Arm program can be transformed in several ways to forbid the outcome $c = 1, d = 0$. In particular, the code of the second thread can be transformed into either of the following two cases, *(i)* $\mathtt{RMW}_{\mathtt{rlx}}(\mathtt{Y}); \mathtt{DMBFF}; \mathtt{LDR}_{\mathtt{Q}}(\mathtt{X})$, and *(ii)* $\mathtt{RMW}_{\mathtt{AL}}(\mathtt{Y}); \mathtt{LDR}_{\mathtt{Q}}(\mathtt{X})$, as each case independently orders the first and last instruction. Such access strengthenings are expected to slow down program execution, which is generally inevitable. However, different transformations might incur different slowdowns. Our algorithm selects a strengthening following a performance cost model, selecting transformation (i) in this case.

Unlike the mapping schemes in Chapters 2 to 4, an important benefit of this approach is that it does not always strengthen Arm's primitives like x86's. Instead, we strengthen accesses only when necessary within their particular program context, benefiting performance. For instance, the final x86 $\mathtt{RMW}$ on the first thread orders stronger than the corresponding Arm $\mathtt{RMW}_{\mathtt{rlx}}$, *in general*. However, *within this partic-*

ular program, the preceding $\mathtt{RMW}_{\mathtt{AL}}$ (on Arm) already orders that subsequent $\mathtt{RMW}_{\mathtt{rlx}}$. Hence, knowing the full program allows judicious strengthening of Arm primitives only when required to preserve the order, while many primitives independently remain weaker than those in the corresponding x86 program.

6.2.2 *Adapted Arm Model*

We use the same Arm model as presented in Section 1.3, except for one additional requirement for our analysis. In addition to those conventional definitions, we enforce Aporf in Arm to avoid $(\mathsf{po} \cup \mathsf{rf})$ cycle in an execution (Lahav et al., 2017; Geeson and Smith, 2024; Lee et al., 2023), which formally is:

$$(\mathsf{po} \cup \mathsf{rf}) \;\text{ is acylic} \tag{Aporf}$$

Finally, the definition of bob we use in this chapter includes the small $\mathtt{CAS}_{\mathtt{AL}}$ fix, *i.e.*, '$\mathsf{bob} \triangleq [\mathsf{codom}(\mathsf{rmw}_{\mathsf{AL}})];\mathsf{po} \cup \ldots$', making the write event of a $\mathtt{CAS}_{\mathtt{AL}}$ instruction order with subsequent events. That corresponds to the model presented in Section 1.3 but differs from the fix we proposed in Chapter 3.

6.3 OPERATIONAL-AXIOMATIC WEAK MEMORY MODEL

We define a combined operational-axiomatic semantics for the weak memory models of x86 and Arm. While an axiomatic model (Section 1.2) defines consistency of an existing execution graph, an operational-axiomatic model describes how to *construct* that graph by executing the program instruction-by-instruction. We thus define the operational-axiomatic model as a Labeled Transition System (LTS) $\mathcal{L} = (\mathcal{S}, \delta)$ where $\mathcal{S}$ is the set of states and δ is a set of labeled transitions. Each state $s \in \mathcal{S}$ is an execution graph $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{rmw} \rangle$ consistent with (axiomatic) model M. Given two states X_1 and X_2, we have a labeled transition $\mathsf{X}_1 \xrightarrow{e} \mathsf{X}_2$ iff both:

- Event e is obtained by executing the next instruction on execution graph X_1, or a pair of read/write events $\langle e_r,\ e_w \rangle$ for a successful $\mathtt{RMW}$ instruction; and
- the result of this execution is the execution graph X_2.

Intuition. Intuitively, we restrict the resulting execution to any extension of the source execution X with e, provided that graph remains well-formed and consistent in its corresponding architecture.

$$\dfrac{\begin{array}{c} \mathsf{E}' = \mathsf{X.E} \cup \{e\} \qquad \mathsf{po}' = \mathsf{X.po} \cup \{(x_{\mathrm{pre}},\ e) \mid x_{\mathrm{pre}} \in \mathsf{X.E} \ \land\ x_{\mathrm{pre}}.\mathsf{tid} = e.\mathsf{tid}\} \\ \mathsf{X.rf} \subseteq \mathsf{rf}' \qquad \mathsf{X.mo} \subseteq \mathsf{mo}' \qquad \mathsf{X.rmw} \subseteq \mathsf{rmw}' \\ \mathsf{X}' = \langle \mathsf{E}', \mathsf{po}', \mathsf{rf}', \mathsf{mo}', \mathsf{rmw}' \rangle \\ \mathsf{X}' \text{ is well-formed} \qquad \mathsf{X}' \text{ is consistent in } M \end{array}}{\mathsf{X} \xrightarrow[M \in \{\mathsf{x86},\mathsf{Arm}\}]{e} \mathsf{X}'}\ \textsc{Intuition}$$

This definition explicitly extends X and po with the new event, while only implicitly extending mo, rf, and rmw to *anything* – axiomatically constrained by well-formedness and consistency. Although this definition axiomatically captures everything we need, it does not state how to *construct* the resulting X′ (and corresponding mo, rf, and rmw relations). Instead, we refine this definition below to avoid explicit well-formedness and consistency checking at every step.

Specifically, we compute the *visible writes* for the reads and write events. When adding an event e, they write to the same location as e, and e can select any to extend from – with mo when e is a write or with rf when e is a read. We consider an RMW instruction to generate two separate read and write events, which are handled differently from non-RMW read/write events.

Visible Writes. When appending a new read/write event e, those visible writes differ between x86 and Arm, because they have different consistency axioms. Their primary difference is that x86 enforces irreflexivity of hb while Arm enforces irreflexivity of ob. In x86, a write w is visible to the new event e if either:

- w and e are concurrent – In this case, w and e appear on different threads *and* they are not ordered by hb. Formally, $(w, e) \notin (\mathsf{hb} \cup \mathsf{po}_{\mathsf{loc}})$; or
- $w \in \mathsf{W}_{e.\mathsf{loc}}$ is an immediate '$[\mathsf{W}_{e.\mathsf{loc}}];(\mathsf{hb} \cup \mathsf{po})$' predecessor of e, which means w is a $(\mathsf{hb} \cup \mathsf{po})$-predecessor of e and there is no other same-location W event $(\mathsf{hb} \cup \mathsf{po})$ in-between.

The intuition is that any such visible write may be observed by the new event e – either by reading from it (rf) or overwriting it (mo). However, there are additional challenges when either e or w originates from an RMW instruction, which we explain below. First, we give the above prose definitions of visible writes *formally*. While the above definitions concern only x86 (with hb), it also mirrors to Arm (with ob), resulting in:

$$\mathsf{VW}_M(\mathsf{X}, e) \triangleq \{w \in \mathsf{X}.\mathsf{W}_{e.\mathsf{loc}} \mid (w, e) \notin \mathsf{X}.(S_M \cup \mathsf{po}_{\mathsf{loc}}) \vee (w, e) \in \mathsf{X}.([\mathsf{W}_{e.\mathsf{loc}}];(S_M \cup \mathsf{po}))_{\mathsf{imm}}\}$$

When M is 'x86', then $S_M = \mathsf{hb}$; when M is 'Arm', then $S_M = \mathsf{ob}$.

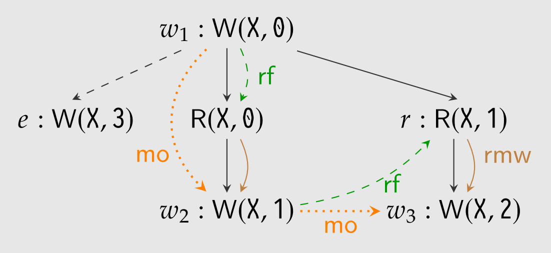
Read. When e is a read events that was *not* produced by a RMWinstruction, its extension simply selects any visible write w and reads-from (rf) it. Formally, we define this extension as:

$$\dfrac{\begin{array}{c}\mathsf{E}' = \mathsf{X}.\mathsf{E} \cup \{e\} \qquad \mathsf{po}' = \mathsf{X}.\mathsf{po} \cup \{(x_{\text{pre}},\ e) \mid x_{\text{pre}} \in \mathsf{X}.\mathsf{E}\ \wedge\ x_{\text{pre}}.\mathsf{tid} = e.\mathsf{tid}\} \\ \mathsf{X}' = \langle \mathsf{E}', \mathsf{po}', \mathsf{X}.\mathsf{rf}, \mathsf{X}.\mathsf{mo}, \mathsf{X}.\mathsf{rmw}\rangle \qquad w \in \mathsf{VW}_M(\mathsf{X}', e) \qquad \mathsf{rf}' = \mathsf{X}.\mathsf{rf} \cup \{(w, e)\}\end{array}}{\mathsf{X} \xrightarrow[M \in \{\mathsf{x86}, \mathsf{Arm}\}]{e} \langle \mathsf{E}', \mathsf{po}', \mathsf{rf}', \mathsf{X}.\mathsf{mo}, \mathsf{X}.\mathsf{rmw}\rangle}\ \text{READ}$$

Unlike our intuitive definition above (INTUITION), we do not include the explicit well-formedness and consistency checks, because we ensure those *by construction* of the resulting execution.

Write. When e is a write event produced by a non-RMW instruction, we can take a similar approach as with reads. We select a preceding visible write to the same location, which e will *overwrite*. Observe that we can insert e in-between two existing mo-related events, where it overwrites the first and is itself overwritten by the second. Although this scenario appears similar to a *read* instruction, it differs because we need to account for the situation where e overwrites an existing RMW-write. To illustrate this situation, consider the following program and corresponding execution:

```
            X = 0;
X = 3; || RMW(X, 0, 1); || RMW(X, 1, 2);
```



To e, all the writes w_1,w_2,w_3 are visible: w_1 is $\mathsf{po}_{\mathsf{locimm}}$-preceding, while w_2 and w_3 are concurrent. Hence, e appears capable of overwriting any of those existing writes. However, for instance, if we decide to overwrite w_2 with e, then e should also occur *after* r, which observed w_2. Then, when e occurs after r, it needs to also occur *after* w_3 to avoid violating Atomicity. The primary observation here is that $(\mathsf{rf};\mathsf{rmw})^+$ chains must be considered as atomic entities, where e can only overwrite the *final* write in such chains – being w_3 in the above example. Formally, we select $w \in \mathsf{VW}_M(\mathsf{X}, e) \setminus \mathsf{dom}(\mathsf{rf};\mathsf{rmw})$, which does not select the starting or intermediate writes in such chains. Unlike with read events – where we modified rf – when e is a write, we modify mo instead, resulting in the following operational-axiomatic rule:

$$\frac{\begin{array}{c}\mathsf{E}' = \mathsf{X.E} \cup \{e\} \qquad \mathsf{po}' = \mathsf{X.po} \cup \{(x_{\mathrm{pre}},\ e) \mid x_{\mathrm{pre}} \in \mathsf{X.E}\ \wedge\ x_{\mathrm{pre}}.\mathsf{tid} = e.\mathsf{tid}\} \\ \mathsf{X}' = \langle \mathsf{E}', \mathsf{po}', \mathsf{X.rf}, \mathsf{X.mo}, \mathsf{X.rmw}\rangle \qquad w \in \mathsf{VW}_M(\mathsf{X}', e) \setminus \mathsf{dom}(\mathsf{X.rf};\mathsf{X.rmw}) \\ \mathsf{mo}' = \mathsf{X.mo} \cup \{(w, e_w)\} \cup \{(w_{\mathrm{pre}}, e_w) \mid (w_{\mathrm{pre}}, w) \in \mathsf{X.mo}\} \\ \cup\{(e_w, w_{\mathrm{post}}) \mid (w, w_{\mathrm{post}}) \in \mathsf{X.mo}\}\end{array}}{\mathsf{X} \xrightarrow[M \in \{\mathsf{x86},\mathsf{Arm}\}]{e} \langle \mathsf{E}', \mathsf{po}', \mathsf{X.rf}, \mathsf{mo}', \mathsf{X.rmw}\rangle}\ \textsc{write}$$

In the construction of mo′, e explicitly "inherits" the mo-edges of w, placing e directly after w in its existing mo chain.

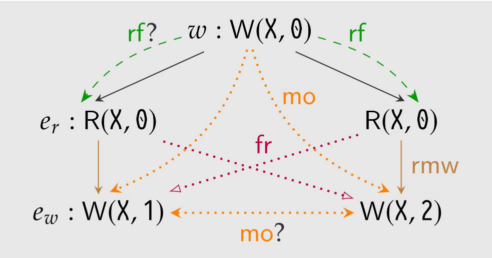
Update (RMW). We handle RMW differently from regular read and write instructions, as RMW behavior is much more subtle. A RMW instruction atomically reads *and* writes. Some RMW variants use the read value to obtain the written value, such as a compare-and-swap operation, which only writes if the read value matches a desired value. Hence, a RMW *fails* whenever that read value does not match, and thus does not generate the rmw relation in the execution. A RMW may also fail whenever another thread modifies the memory location in-between atomically reading and writing (*i.e.*, violating Atomicity in Section 1.3). When specifying the operational-axiomatic semantics, we need to account for all those cases, *in addition to* the concerns we listed for regular read and write instructions.

In the semantics, we must describe the validity of both the success and failure steps. In some executions, the `RMW` necessarily fails, *e.g.*, when the read value mismatches. In other executions, the `RMW` necessarily succeeds, *e.g.*, when the read value matches and no other thread can overwrite its memory location. In most other cases, both success and failure are possible, *e.g.*, when the read value matches but another thread does not necessarily (but possibly) modify the location in memory in-between the atomic read and write. We thus describe those cases separately, where success simultaneously adds a $\langle e_r, e_w \rangle$ pair of read and write events, while failure includes only the read e_r.

For the success case, e_r reads from a pre-existing write w, like we saw for non-`RMW` reads. Suppose we thus select $w \in \mathsf{VW}(\ldots)$. If *another* successful `RMW` had already read from w, that other `RMW` necessarily overwrites w. Hence, if e also were to read-from (rf) w, its corresponding write e_w would violate atomicity, which the following program and corresponding execution demonstrates:

```
          X = 0;
RMW(X,0,1); || RMW(X,0,2);
```

The `RMW`s cannot both be successful in x86 and Arm



Both `RMW` writes are mo-after w, but must also order with each other, in either direction – violating Atomicity in both cases. Hence, a `RMW`-read *cannot* read from a write event (w) that another successful `RMW` had already read from. The `RMW` can only succeed when selecting $w \in \mathsf{VW}_M(\mathsf{X}, e) \setminus \mathsf{dom}(\mathsf{rf};\mathsf{rmw})$. Then, we simply add both the e_r and e_w events to the execution simultaneously, while including them in rf, mo, and rmw:

$$\frac{\begin{array}{c}\mathsf{E}' = \mathsf{X.E} \cup \{e\} \qquad \mathsf{po}' = \mathsf{X.po} \cup \{(x_{\text{pre}},\ e_r) \mid x_{\text{pre}} \in \mathsf{X.E}\ \wedge\ x_{\text{pre}}.\mathsf{tid} = e_r.\mathsf{tid}\} \\ \mathsf{po}'' = \mathsf{po}' \cup \{(x_{\text{pre}},\ e_w) \mid x_{\text{pre}} \in \mathsf{X.E}\ \wedge\ x_{\text{pre}}.\mathsf{tid} = e_w.\mathsf{tid}\} \cup \{(e_r, e_w)\} \\ \mathsf{X}' = \langle \mathsf{E}', \mathsf{po}'', \mathsf{X.rf}, \mathsf{X.mo}, \mathsf{X.rmw} \rangle \qquad w \in \mathsf{VW}_M(\mathsf{X}', e_r) \setminus \mathsf{dom}(\mathsf{X.rf};\mathsf{X.rmw}) \\ \mathsf{rf}' = \mathsf{X.rf} \cup \{(w, e_r)\} \qquad \mathsf{rmw}' = \mathsf{X.rmw} \cup \{(e_r, e_w)\} \\ \mathsf{mo}' = \mathsf{X.mo} \cup \{(w, e_w)\} \cup \{(w_{\text{pre}}, e_w) \mid (w_{\text{pre}}, w) \in \mathsf{X.mo}\} \\ \cup \{(e_w, w_{\text{post}}) \mid (w, w_{\text{post}}) \in \mathsf{X.mo}\}\end{array}}{\mathsf{X} \xrightarrow[M \in \{\mathsf{x86}, \mathsf{Arm}\}]{\langle e_r,\ e_w \rangle} \langle \mathsf{E}', \mathsf{po}'', \mathsf{rf}', \mathsf{mo}', \mathsf{rmw}' \rangle}\ \text{RMW-OK}$$

Although this rule appears complex, its definition merely combine those of READ and WRITE, with some minor additions. Here, po′ includes e_r into the execution's po, while po″ also includes e_w. Finally, rf′ ensures e_r reads-from the correct write, which is e_w's predecessor in mo′.

In contrast, a failing RMW largely mirrors the definition of a regular READ:

$$\dfrac{\begin{array}{c} \mathsf{E}' = \mathsf{X.E} \cup \{e_r\} \qquad \mathsf{po}' = \mathsf{X.po} \cup \{(x_{\text{pre}}, e_r) \mid x_{\text{pre}} \in \mathsf{X.E} \ \wedge \ x_{\text{pre}}.\mathsf{tid} = e_r.\mathsf{tid}\} \\ \mathsf{X}' = \langle \mathsf{E}', \mathsf{po}', \mathsf{X.rf}, \mathsf{X.mo}, \mathsf{X.rmw}\rangle \qquad w \in \mathsf{VW}_M(\mathsf{X}', e_r) \\ e_r.\mathsf{desired} \neq w.\mathsf{val} \ \vee \ \exists v.\mathsf{mo}(w, v) \qquad \mathsf{rf}' = \mathsf{X.rf} \cup \{(w, e_r)\} \end{array}}{\mathsf{X} \xrightarrow[M\in\{\mathsf{x86},\mathsf{Arm}\}]{e_r} \langle \mathsf{E}', \mathsf{po}', \mathsf{rf}', \mathsf{X.mo}, \mathsf{X.rmw}\rangle} \ \text{RMW-FAIL}$$

The only difference between a regular READ and a failed RMW-read (RMW-FAIL) is the additional failure condition. After all, a RMW can only fail if the desired value mismatches – *i.e.*, when $e_r.\mathsf{desired} \neq w.\mathsf{val}$ – or whenever there exists another write v that may occur in-between the atomic read and write – *i.e.*, when $\exists v.\mathsf{mo}(w, v)$.

Fence. Whenever e is a fence, inserting it into the execution is trivial because it does not modify any of the rf, mo, rmw relations. The corresponding rule is:

$$\dfrac{\mathsf{E}' = \mathsf{X.E} \cup \{e\} \qquad \mathsf{po}' = \mathsf{X.po} \cup \{(x_{\text{pre}}, e) \mid x_{\text{pre}} \in \mathsf{X.E} \ \wedge \ x_{\text{pre}}.\mathsf{tid} = e.\mathsf{tid}\}}{\mathsf{X} \xrightarrow[M\in\{\mathsf{x86},\mathsf{Arm}\}]{e} \langle \mathsf{E}', \mathsf{po}', \mathsf{rf}, \mathsf{mo}, \mathsf{rmw}\rangle} \ \text{FENCE}$$

MPMC-Queue example (Figure 26). The execution in Figure 27 demonstrates the robustness violation of the program in Figure 26b. The execution is forbidden in x86 as $\mathsf{xppo}(m, n)$ holds, creating a hb cycle, whereas Arm allows this execution, as m and n are unordered. Now consider the strengthened program with $\mathtt{RMW}_{\mathtt{rlx}}(Y) \rightsquigarrow \mathtt{RMW}_{\mathtt{A}}$ in the second thread and its execution $c = 1, d = 0$ in (b). Then, $\mathsf{bob}(m, n)$ holds, resulting in an ob cycle that forbids this execution in Arm as well.

6.4 ROBUSTNESS ANALYSIS

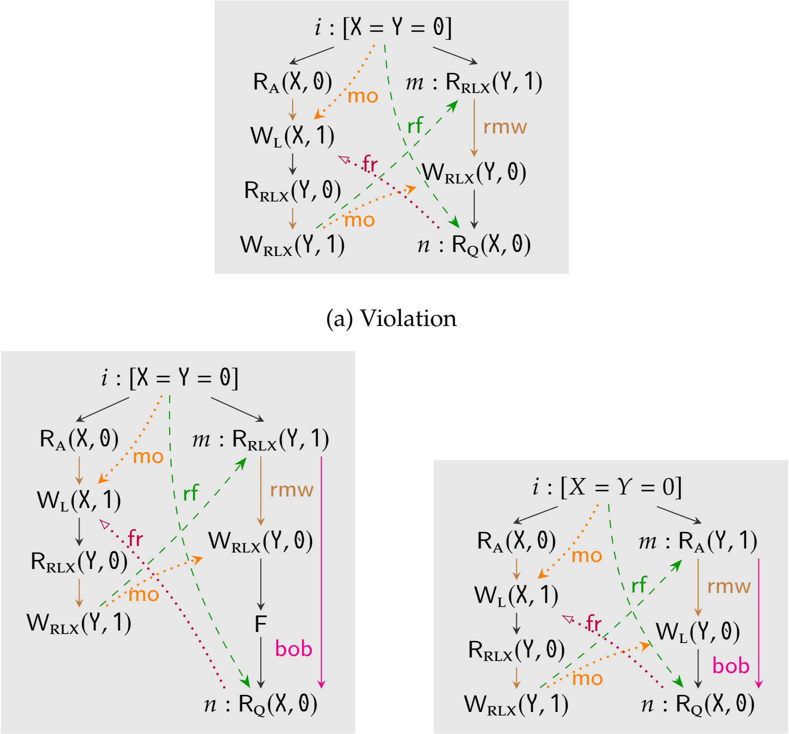
In this section, we check if an execution of a program is allowed in Arm but not in x86. We formulate this question as a robustness analysis problem and propose our approach.

6.4.1 *Robustness Violation*

A program violates (x86, Arm)-robustness if the program has an Arm-consistent execution which is not x86-consistent (*i.e.*, it violates Definition 3).

Example. The executions in Figures 28a and 28b demonstrate robustness violations of the MP and 2+2W programs, respectively, while adding event e_4.

$$\begin{array}{l||l} \mathtt{X = 1;} & a\ \mathtt{= Y;} \\ \mathtt{Y = 1;} & b\ \mathtt{= X;} \end{array} \ \text{(MP)} \qquad\qquad \begin{array}{l||l} \mathtt{X = 1;} & \mathtt{Y = 1;} \\ \mathtt{Y = 2;} & \mathtt{X = 2;} \end{array} \ \text{(2+2W)}$$

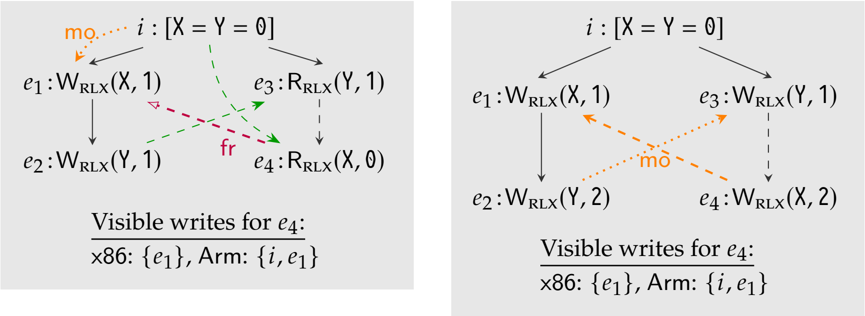


(a) Violation

(b) Fix (i)

(c) Fix (ii)

Figure 27: MPMC-Queue (x86, Arm)-robustness violation from Figure 26 and the possible fixes (all in Arm).



(a)

(b)

Figure 28: Examples of robustness violation steps. For both examples, an *additional* write is visible in Arm. That write is invisible on x86 because e_1 is hb-intermediate between i and e_4 (separately in either example).

In Figure 28a, we append the read event e_4 to the graph consisting of $\{i, e_1, e_2, e_3\}$. In x86, the only write visible to e_4 is e_1. Particularly, the initial write in i is not visible to e_4, because e_1 is hb-intermediate between them. On the other hand, in Arm the visible writes are $\{i, e_1\}$ as e_1 is *not* an ob-intermediate between i and e_4. Hence, on Arm, we can insert e_4 in-between i and e_1, producing both $\mathsf{mo}(i, e_4)$ and $\mathsf{mo}(e_4, e_1)$. However, that is impossible on x86. This mismatch, produced from additional visible writes on Arm, results in the robustness violation.

Figure 28b demonstrates a robustness violation without read events. After appending write event e_4 to the graph consisting of $\{i, e_1, e_2, e_3\}$, the sets of writes visible to e_4 are $\{e_1\}$ in x86 and $\{i, e_1\}$ in Arm. These mismatch since e_1 is a hb-predecessor in x86 (thus making i invisible) but is not a ob-predecessor in Arm. In this case it is possible to create $\mathsf{mo}(i, e_4)$ and $\mathsf{mo}(e_4, e_1)$ in Arm, but not in x86.

6.4.2 *Analysis Algorithm*

The (x86, Arm)-robustness analysis algorithm constructs an execution following the operational-axiomatic model in Section 6.3 such that the execution remains both x86 and Arm consistent. If an execution's construction steps diverge, the algorithm raises a robustness violation. For efficient analysis, we keep track of the coherence order on the same-location events and global orders for both models.

- For x86, we only keep track of hb ordering.
- For Arm, we combine the ob and the coherence ordering on same-location events to define coherence-ordered-by: $\mathsf{cob} \triangleq (\mathsf{ob} \cup [\mathsf{R}];\mathsf{po}_{\mathsf{loc}};[\mathsf{R} \cup \mathsf{W}] \cup [\mathsf{W}];\mathsf{po}_{\mathsf{loc}};[\mathsf{W}])^+$.

Algorithm 1: (x86, Arm)-robustness analysis.

```
Input: execution $X = \langle E, po, rf, mo, rmw, \langle hb, cob \rangle \rangle$ and event $e$ in thread $t$.
case $e \in R \lor e \in W$ do
    let $X_1 \leftarrow \mathsf{append}(X, t, e)$;
    let $X_2 \leftarrow \mathsf{updateIntraOrder}_{\{x86,Arm\}}(X_1, t, e)$;
    let $\langle V_x, V_a \rangle \leftarrow \mathsf{VW}_{\{x86,Arm\}}(X_2, e)$;
    if $(V_a \setminus V_x) \neq \emptyset$ then
        return $\langle X_2, (V_a \setminus V_x),\ e \rangle$;
case $e \in R$ do
    let $w \leftarrow \mathsf{random}(V_a)$;
    let $X' \leftarrow X_2[\mathsf{rf} \mapsto X_2.\mathsf{rf} \cup \{(w, e)\}]$;
case $e \in W$ do
    let $X_3 \leftarrow \mathsf{updateMO}(X_2, e)$;
    let $X' \leftarrow \mathsf{updateIfRMW}(X_3, e)$;
let $X'' \leftarrow \mathsf{updateOrd}_{\{x86,Arm\}}(X',\ e)$;
```

Full Algorithm. The algorithm consists of several components, which we explain top-down. Algorithm 1 performs the (x86, Arm)-robustness analysis for an execution step. The large helper functions are included below (Algorithms 2, 4 and 5), while we describe the smaller helper functions within the text. Given an execution X with additional orderings and a newly appended event e in the thread t the analysis steps are as follows.

- (Lines 1 to 6) When e is either a read or write event – possibly originating from `RMW` – we append it to the execution. Line 2 appends event e to the thread t, while on line 3, updateIntraOrder(...) updates the intra-thread hb and cob orderings in x86 and Arm, respectively, within thread t. Next, line 4 computes the visible writes in x86 and Arm by VW(...), corresponding to the operational-axiomatic semantics (Section 6.3). Line 5 detects whether more writes are visible in Arm than in x86, reporting a robustness violation if so, returning these additional writes on line 6 (which are later fixed – Section 6.5.1). Otherwise, the event e behaves similarly in x86 and Arm, and can be added to the execution graph. As read and write events affect the execution differently, we handle them separately below
- (Lines 7 to 9) When e is a read event, we randomly select a visible write w on line 8, from which we append the corresponding rf edge on line 9.
- (Lines 10 to 12) When e is a write, we invoke updateMO(...) to add e after a random write event $w \in V_a$. Unlike the semantics (*i.e.*, Write – Section 6.3), we select $w \in$ VW(...) instead of $w \in$ VW(...) $\setminus$ dom(rf;rmw). If we have selected $w \in$ dom(rf;rmw), we place it either before or after the entire rf;rmw-chain with updateMO(...), which we explain in more detail below (Algorithm 2). If e was generated by a `RMW` instruction, we add an rmw edge from its corresponding R event to e (on line 12).
- (Line 13) Finally, after constructing a well-formed execution X′, we update our view of hb and cob with updateOrd(...).

A step either performs a state-transition by updating the execution graph or reports the first robustness violation.

UpdateMO. Our updateMO function, which we used above in Algorithm 1, places the new write event e behind an existing write $w \in V$ – we include it in Algorithm 2. Particularly, it ensures mo remains *total* per location, by adding mo-edges between e and all existing writes – in either direction. The loops and conditions on lines 2-4 iterate over all existing same-location write events.

Crucially, we must specifically consider the scenario where w is on a rf;rmw-chain, as e can only be before or after the entire chain – never in the middle, as that would violate Atomicity. Whenever w is on such a chain, we specifically take the start w_{start} and end w_{end} of that chain (lines 5-6); When not on a chain, then $w{=}w_{\text{start}}{=}w_{\text{end}}$. When e is hb/cob-after w (or its entire rf;rmw-chain), then the if-case on lines 7-8 places e mo-after w. When e is hb/cob-before w (or its entire rf;rmw-chain), then the elseif-case on lines 9-10 places e mo-before w. Finally, when e is hb/cob-concurrent with w, we can attach the mo edge in either direction on line 12.

Note that those updated orders may restrict the mo-order of the writes in the next loop iterations, as the created mo edges are included in hb/cob on lines 13-16. After iterating over all writes, e connects to all prior same-location writes, while mo is again total per location.

Algorithm 2: $\mathsf{updateMO}_M()$: Update the mo relation.

Input: Execution graph $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{rmw} \rangle$, a new write event $e = \mathsf{W}(t, x)$.
If $M = \mathsf{x86}$ then $S_M = \mathsf{hb}$ and if $M = \mathsf{Arm}$ then $S_M = \mathsf{cob}$.

```
1  let $\mathsf{X}' \leftarrow \mathsf{X}$;
2  foreach $t \in Threads$ do
3      foreach $w \in \mathsf{events}(t)$ do
4          if $w \in \mathsf{W}_x$ then
5              let $w_{\mathsf{start}} \leftarrow \mathsf{rmwChainStart}(w)$;  -- equals $w$ when not on a rf;rmw-chain
6              let $w_{\mathsf{end}} \leftarrow \mathsf{rmwChainEnd}(w)$;    -- equals $w$ when not on a rf;rmw-chain
7              if $(w_{start}, e) \in \mathsf{X}'.S_M$ then
8                  let $\mathsf{X}'.\mathsf{mo} \leftarrow \mathsf{X}'.\mathsf{mo} \cup \{(w, e)\}$;
9              else if $(e, w_{end}) \in \mathsf{X}'.S_M$ then
10                 let $\mathsf{X}'.\mathsf{mo} \leftarrow \mathsf{X}'.\mathsf{mo} \cup \{(e, w)\}$;
11             else
12                 let $\mathsf{X}'.\mathsf{mo} \leftarrow \mathsf{X}'.\mathsf{mo} \cup \mathsf{random}(\{(w, e), (e, w)\})$;
13             if $M = \mathsf{x86}$ then
14                 $\mathsf{updateXHB}(\mathsf{X}'.\mathsf{hb}, e)$;
15             else
16                 $\mathsf{updateCOB}(\mathsf{X}'.\mathsf{cob}, e)$;
17 return $\mathsf{X}'$;
```

Compute visible writes. Here, we give the algorithm to *compute* the visible writes $\mathsf{VW}(\ldots)$, which we already included declaratively with the operational-axiomatic semantics (Section 6.3). Algorithm 3 computes those visible writes under the x86 and Arm model for an event e that is appended to an execution graph X, on thread t and location x. We accumulate the visible writes in two phases. First, on lines 1-11, we accumulate a set of writes VW_1 that are possibly visible to the appended event e. Second, on lines 12-20, we filter VW_1 to those that are *actually* visible, collected in VW.

In the first phase, we traverse the events in each thread t in reverse to identify the latest write or RMW event e' on the same location as e (lines 3-11). If a write event e' on the same location x is hb in x86 or cob in Arm, or immediate $\mathsf{po}_{\mathsf{loc}}$ predecessor of event e, then we accumulate e' in VW_1 (lines 5-8). Note that we do not need to look further in the thread t, as only one event is immediately hb/cob/$\mathsf{po}_{\mathsf{loc}}$-preceding – hence the break on line 8. Otherwise, e' is hb/cob-concurrent to e, making it possibly visible (lines 9-11)

In the second phase (lines 12-20), we iterate over the accumulated writes in $w \in \mathsf{VW}_1$ and check whether it is visible to e. In particular, when another write $w' \in \mathsf{VW}_1$ is in-between w and e (by hb/cob), then w is *not* visible (lines 15-18).

Algorithm 3: $\mathsf{VW}_M()$: Compute Visible-Writes in $M \in \{\mathsf{x86}, \mathsf{Arm}\}$.

```
Input: Execution graph $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo} \rangle$ and a new event $e = \mathsf{R}(t, x)$.
If $M$ = x86 then $S_M$ = hb and if $M$ = Arm then $S_M$ = cob.
let $\mathsf{VW}_1 \leftarrow \emptyset$;
foreach $t \in Threads$ do
    foreach $e' \in \mathsf{events}(t).\mathsf{rev}()$ do
        if $e' \in \mathsf{W}_x$ then
            if $(e', e) \in \mathsf{X}.S_M \vee t = e.\mathsf{tid}$ then
                // $S_M \cup \mathsf{po}_{\mathsf{loc}}$-preceding on the same thread
                let $\mathsf{VW}_1 \leftarrow \mathsf{VW}_1 \cup \{e'\}$;
                break;
            else
                // Concurrent
                let $\mathsf{VW}_1 \leftarrow \mathsf{VW}_1 \cup \{e'\}$;
let $\mathsf{VW} \leftarrow \emptyset$ ;
foreach $w \in \mathsf{VW}_1$ do
    let $visible \leftarrow \top$ ;
    foreach $w' \in \mathsf{VW}_1 \wedge w' \neq w$ do
        if $(w, w') \in \mathsf{X}.S_M$ then
            let $visible \leftarrow \bot$ ;
            break;
    if $visible$ then
        let $\mathsf{VW} \leftarrow \mathsf{VW} \cup \{w\}$;
return $\mathsf{VW}$;
```

6.4.3 *Arm Relations*

In Arm, we update cob in two ways. We update the intra-thread order (*i.e.,*cob within po) with $\mathsf{updateIntraOrder}_{\mathsf{Arm}}$ – used in Algorithm 1 – while we updated the external cob in updateMO (Algorithm 2). We explain these updates to cob separately.

Intra-thread. Given an event e, we compute the intra-thread cob relation in Algorithm 4 as follows. Line 1 computes the aob case of cob. Lines 2 to 9 computes the bob case of cob targeting e. We factored out the first three cases (lines 2-4), each of which starts at a specific event v. Hence, we can obtain that specific event with $\mathsf{lastEv}(\ldots)$, which returns the last event of a specific type in the given thread. Those do not need to look further backwards, as any preceding events v' of the same type would already be ordered before it (*i.e.,* $\mathsf{cob}(v', v)$)

The final bob cases (lines 5-9) order *through* an intermediate event with any preceding event. For instance, with an intermediate F_{RM}, we must cob-order *all* R events preceding the fence before e. When implementing the algorithm, one could memoize those read events at fence f and prune transitive redundancies; *e.g.*, if $\mathsf{cob}(x, f)$, $\mathsf{cob}(y, f)$, and $\mathsf{cob}(x, y)$ we need not memoize x at f when it already knows y.

External. Given an event e, we extend the cob in Algorithm 5 as follows: When e is a write event, lines 1-5 add incoming fre edges from all other threads into cob. When e is a read event, lines 6-11 either includes rfe into cob (line 9) or the special '$[\mathsf{codom}(\mathsf{rmw})];\mathsf{rfi};[\mathsf{R_A} \cup \mathsf{R_Q}]$' case of aob (line 11). Finally, on line 12, $\mathsf{propagate}(\ldots)$ propagates the cob edges to and from e to the other events in the graph, establishing cob's global transitive closure.

```
Algorithm 4: updateIntraOrder_Arm(): Update intra-thread cob in Arm.
  Input: Execution $X = \langle E, po, rf, mo, rmw, cob \rangle$ and event $e$ appended in thread $t$.
  let cob ← $cob \cup \{(v, e)\}$ where $rmw(v, e)$;                                  // aob
  let cob ← $cob \cup \{(v, e)\}$ where $v = lastEv(R_A \cup R_Q, t)$;                  // bob (2)
  let cob ← $cob \cup \{(v, e)\}$ where $v = lastEv(W_L, t) \wedge e \in R_A$;           // bob (4)
  let cob ← $cob \cup \{(v, e)\}$ where $v = lastEv(codom([R_A];rmw;[W_L]), t)$;       // bob (8)
  let cob ← $cob \cup \{(v, e)\}$ where $v{,}e \in R \cup W \wedge \exists f \in F_{MM}.[po(v, f) \wedge po(f, e)]$;   // bob (1)
  let cob ← $cob \cup \{(v, e)\}$ where $v \in R \wedge e \in R \cup W \wedge \exists f \in F_{RM}.[po(v,f) \wedge po(f,e)]$;  // bob (3)
  let cob ← $cob \cup \{(v, e)\}$ where $v, e \in W \wedge \exists f \in F_{WW}.[po(v, f) \wedge po(f, e)]$;   // bob (5)
  let cob ← $cob \cup \{(v, e)\}$ where $\langle v, e \rangle \in po \wedge e \in W_L$;   // bob (6)
  let cob ← $cob \cup \{(v, e)\}$ where $\exists w \in W_L.[po(v, w) \wedge po(w, e)] \wedge e \in W$;   // bob (7)
```

```
Algorithm 5: updateCOB(): Update global Arm cob relation.
   Input: Execution $X = \langle E, po, rf, mo, rmw, cob \rangle$ and event $e$ appended in thread $t$.
1  case $e \in W$ do
2      foreach $t' \in (Threads \setminus \{t\})$ do
3          let $r \leftarrow lastEv(R_{e.loc}, t')$;
4          if $(e, r) \notin cob$ then
5              let cob ← $cob \cup \{(r, e)\}$ // fre edge ;
6  case $e \in R$ do
7      let $w_{rf} = rf^{-1}(e)$ ;
8      if $w_{rf}.tid \neq e.tid$ then
9          let cob ← $cob \cup \{(w_{rf}, e)\}$ // rfe edge;
10     else if $w_{rf} \in codom(rmw) \wedge e.ord \in (R_A \cup R_Q)$ then
11         let cob ← $cob \cup \{(w_{rf}, e)\}$ // aob edge;
12 propagate(cob, $e$);
```

6.4.4 x86 Relations

The approach for x86 is much easier, as its model has fewer and simpler cases. Particularly, we update the intra-thread relations of hb (*i.e.*, xppo) and its external cases.

Intra-thread. For each thread we *memoize* the last read (LR) and last write (LW) within the thread, from which we extend the xppo edges. Then, given an event e we update the xppo orders as shown in Algorithm 6. Contrasting Arm's cob, xppo edges may directly originate from or target fences, making it easier to add the edges. When e is a read event, Lines 1 to 3 add edges from any preceding read event to e, and update LR. Observe that write-read pairs are unordered in x86 (by xppo). Whenever e is either a write or fence event, we add edges from *any* preceding event to e in 4-8. Note that when e is fence, we still mark it as "last read" and "last write" because it has in incoming xppo-edge from the actual last preceding read and write events; This approach allows us to preserve the order of write-read pairs (by xppo^+) when there is an intermediate fence.

Algorithm 6: $\text{updateIntraOrder}_{\text{x86}}()$: Update xppo in x86 execution.

```
   Input: Thread $t$ with newly appended event $e$.
1  case $e \in \mathsf{R}$ do
2      let xppo ← xppo ∪ $\{(t.\mathsf{LR}, e)\}$;
3      let $t.\mathsf{LR} \leftarrow e$ ;
4  case $e \in \mathsf{W} \cup \mathsf{F}$ do
5      let xppo ← xppo ∪ $\{(t.\mathsf{LW}, e), (t.\mathsf{LR}, e)\}$ ;
6      let $t.\mathsf{LW} \leftarrow e$;
7      case $e \in \mathsf{F}$ do
8          let $t.\mathsf{LR} \leftarrow e$
```

External. Finally, to update the global hb order (used in updateMO) for a new event e, we add the incoming fre and rfe from other threads, shown in Algorithm 7. The approach resembles Arm's updateCOB (Algorithm 5). When e is a write, Lines 1 to 5 include fre edges into hb from reads on all other threads. Note that lastRead$(\ldots)$ retrieves the last read on a particular location for a specific thread – it thus differs from LR, containing any event that observed the last read to any location. When e is a read, Lines 6 to 9 include the rfe edge into hb. Finally, propagate$(\ldots)$ on line 10 propagates the hb edges to and from e to the other events in the execution graph, establishing hb's global transitive closure.

6.5 ROBUSTNESS ENFORCEMENT

Algorithm 1 returns a robustness violation $\langle \mathsf{X}, W, e\rangle$. In this case there exists an execution graph X_0 which is consistent in both x86 and Arm and X po-extends graph X_0 by event e. In X, W is the set of writes visible in Arm but not in x86 to event e. We enforce robustness for the event pair (w, e) where w is selected from the set W. In this case, (w, e) is in X.hb relation in x86, but not in X.cob relation in Arm. It implies there exists a path from w to e consisting of a nonempty set of X.po and

Algorithm 7: updateXHB(): Update x86 global hb relation.

```
Input: hb relation and event $e$
case $e \in \mathsf{W}(t, x)$ do
    foreach $t' \in Threads \wedge t' \neq e.tid$ do
        let $r \leftarrow$ lastRead($t', x$) ;
        if $(e, r) \notin$ hb then
            let hb $\leftarrow$ hb $\cup \{(r, e)\}$   // fre edge;
case $e \in \mathsf{R}(t, x)$ do
    let $w_{\mathsf{rf}} = \mathsf{rf}^{-1}(e)$ ;
    if $w_{\mathsf{rf}}$.tid $\neq e$.tid then
        let hb $\leftarrow$ hb $\cup \{(w_{\mathsf{rf}}, e)\}$   // rfe edge
propagate(hb, $e$);
```

(X.rfe ∪ X.fre ∪ X.moe) edges that enforces X.hb but does not enforce X.cob. It results due to the *weak* po-*edges* on this path where the po is unordered in Arm but ordered in x86, since (X.rfe ∪ X.fre ∪ X.moe)-enforce hb and ob orders in both x86 and Arm respectively.

Fixing the violation. Given such a path $w \rightarrow e_1 \rightarrow e_2 \ldots \rightarrow e$, an enforcement strengthens the *weak* po-*edges* on that path to also order w and e in Arm. To this end, we strengthen the memory order of certain accesses involved in the weak po-edges or insert fences. Each weak edge can be strengthened in multiple ways based on its context. Given all possible strengthenings with respective costs following a given cost model, an *optimal robustness enforcement* strategy selects the transformation strategy that incurs minimal cost among all cases.

6.5.1 *Strengthening Algorithm*

Algorithm 8 strengthens accesses, eliminating a violating path from being reported again.

- The outer loop (line 2-17) iterates over all (po ∪ rfe ∪ moe ∪ fre) paths in X between w and e, looking for the cheapest path to fix. Observe that the number of paths (obtained by allPaths) may be *exponential* in the trace/graph length. We prune that space by first identifying events that occur on *any* path (in $O(n)$) and then enumerate only the paths including those. While this enumeration is still exponential, violating events are often close together (<100 events apart), and this approach is sufficient to analyze our benchmark traces.
- Given a path, we extract its maximal po-edges (line 3) $a \longrightarrow\!\!\!\rightarrow^* b$, which are maximal sequences of po-related events, where a and b are on different memory locations. Suppose our edge is of the form $(x_{\text{start}}, xs_{\text{mid}}, x_{\text{end}})$, where $x_{\text{start}} = a$, $x_{\text{end}} = b$, and there is an intermediate sequence of po-related events xs_{mid}. In the inner loop, we fix this *one* edge $a \longrightarrow\!\!\!\rightarrow^* b$ (line 7-12). However, to fix the entire path, we strengthen

Algorithm 8: Robustness enforcement.

```
   Input: Program $\mathbb{P}$, Execution X, Event $w$, Event $e$
1  let bestFix ← $\varepsilon$, bestFixCost ← $\infty$;
2  foreach $p \in$ allPaths(X, $w$, $e$) do
3      let pos ← extractPOs($p$);
4      let pathFixes ← $\emptyset$, pathFixCost ← 0;
5      foreach ($x_{start}$, $xs_{mid}$, $x_{end}$) ∈ pos do
6          let poFix ← $\varepsilon$, poFixCost ← $\infty$;
7          foreach $x_{mid} \in \{\varepsilon\} \cup xs_{mid}$ do
8              let fixOpts ← strengthen($x_{\text{start}}$, $x_{\text{mid}}$, $x_{\text{end}}$);
9              foreach $f \in$ fixOpts do
10                 if cost($f$, $\mathbb{P}$) < poFixCost then
11                     poFix ← $f$;
12                     poFixCost ← cost($f$, $\mathbb{P}$);
13         pathFixes ← pathFixes ∪ {poFix};
14         pathFixCost ← pathFixCost + poFixCost;
15     if pathFixCost < bestFixCost then
16         bestFix ← pathFixes;
17         bestFixCost ← pathFixCost;
```

all maximal po edges on that path (line 5-14).

- An individual edge $a \rightarrow\rightarrow^{*} b$, may be fixed by multiple ways. For instance, for an unordered $\mathsf{R}_{\text{RLX}} - \mathsf{W}_{\text{RLX}}$ pair, we could *either* strengthen the R to R_{A} *or* the W to W_{L}. strengthen (line 8) returns *all* options that apply:

$$\begin{array}{l}
(\mathtt{LDR},\_,\_) \rightarrow (\mathtt{LDR}_{\mathsf{Q}},\_,\_)\\
(\_,\_,\mathtt{STR}) \rightarrow (\_,\_,\mathtt{STR}_{\mathsf{L}})\\
(\mathtt{STR},\_,\mathtt{LDR}) \rightarrow (\mathtt{STR}_{\mathsf{L}},\_,\mathtt{LDR}_{\mathsf{A}})\\
(\_,\mathsf{F}_{x},\_) \rightarrow (\_,\mathsf{F},\_)\\
(\_,\mathtt{STR},\mathtt{LDR}) \rightarrow (\_,\mathtt{STR}_{\mathsf{L}},\mathtt{LDR}_{\mathsf{A}})\\
(\mathtt{STR},\mathtt{LDR},\_) \rightarrow (\mathtt{STR}_{\mathsf{L}},\mathtt{LDR}_{\mathsf{A}},\_)
\end{array}$$

$$\begin{array}{l}
(\_,\mathtt{RMW},\_) \rightarrow (\_,\mathtt{RMW}_{\mathsf{AL}},\_)\\
(\mathtt{RMW}(\mathsf{R}),\_,\_) \rightarrow (\mathtt{RMW}_{\mathsf{A}},\_,\_)\\
(\mathtt{RMW},\_,\_) \rightarrow (\mathtt{RMW}_{\mathsf{AL}},\_,\_)\\
(\_,\_,\mathtt{RMW}(\mathsf{W})) \rightarrow (\_,\_,\mathtt{RMW}_{\mathsf{L}})\\
(\_,\_,\mathtt{RMW}) \rightarrow (\_,\_,\mathtt{RMW}_{\mathsf{AL}})
\end{array}$$

The instructions on the left-hand side imply any memory order. Applying *any one* of these options put the po-edge in Arm's bob, thus strengthening it in Arm. Note that a path sometimes only considers a RMW's R or W event; *e.g.*, an rf reaching a RMW targets its R, which we denote with RMW(R). The middle event represents an intermediate event (*e.g.*, a F or RMW), whose strengthening may also order the chain (line 7).

- Among all options, we select the cheapest fix for the chain (line 10). Our cost model follows an intuitive hierarchy between memory-orders and instruction rlx < {A, L} < AL and {LDR, STR} < RMW; That is, for instance, strengthening mm is more expensive than rlx, and an RMW is more expensive than strengthening a LDR. Additionally,

we consider the cost *within context* of the instruction in the program. For instance, strengthening an $\texttt{RMW}_{\texttt{A}}$ to $\texttt{RMW}_{\texttt{AL}}$ is cheaper than strengthening an $\texttt{RMW}_{\texttt{rlx}}$ to $\texttt{RMW}_{\texttt{AL}}$.
- Finally, after fixing all po-edges, we found the cheapest way of fixing the full path. If it is the cheapest path to fix overall (line 15-17), we select the current path.

This algorithm can fix the errors in Figure 27 (MPMC-Queue) and Figure 28 (both a and b) by establishing the bob relation.

6.6 EXPERIMENTAL EVALUATION

We implemented the robustness analysis on C11Tester by Luo and Demsky (2021), a state-of-the-art C/C++ testing tool. C11Tester constructs execution graph of C/C++ programs. At each step, it computes the relations and orders for the C11 concurrency model by Lahav and Margalit (2019). In contrast, our Origami computes those of the x86 and Arm models while constructing the execution graph for the analyses. Origami keeps a *vector clock* (Lamport, 2019) for every event, containing its last observed read and write events *per-memory location*. When event e_2 *observes* another event e_1 (*e.g.*, with updateMO or updateOrd), e_2's Arm vector clock includes e_1 and everything observed by e_1. Hence, we can obtain mo explicitly by following same-location writes in the vector clocks.

Experimental Setup. We compare against state-of-the-art x86 to Arm translation approaches: Fency by Chakraborty (2021), and our mapping schemes from (Chapters 2 to 4). Fency is a static analysis tool for checking and enforcing robustness for several weak memory models, including between x86 and Arm (Chakraborty, 2021). We compare with Origami on their respective benchmarks, where we compare the number of strengthenings and the performance of the generated Arm programs on an AWS `c7gd.metal` instance (with a 64-core Graviton3 CPU). We fix programs until Origami finds no more violations for 1,000 runs. While this may not find all errors in general, we expect it finds all realistic errors for our benchmarks.

The C11Tester and Fency benchmarks are popular concurrent data structures and algorithms with C/C++ concurrency primitives. Lasagne used a subset of the Phoenix benchmark suite Ranger et al. (2007), consisting of shared memory MapReduce programs, widely used to benchmark concurrent executions.

6.6.1 *Research Questions*

We address the following research questions to evaluate our approach:
- **RQ1** Can Origami identify program executions in the C11Tester benchmarks that violate (x86, Arm)-robustness and fix them?
- **RQ2** How does Origami compare against the static robustness analysis and enforcement techniques in Fency?
- **RQ3** How effective is Origami in translating concurrent programs written for x86 to Arm compared to Risotto's mapping scheme?

name	|trace|	#
chase-lev-deque	62	0
spsc-queue	19	2
barrier	53	0
dekker	35	0

name	|trace|	#
mcs-lock	58	0
mpmc-queue	47	3
ms-queue-tsan	116	6
linuxrwlocks	26	0

Figure 29: C11Tester Benchmarks. Trace lengths of the final successful execution are averaged over 1,000 runs. # is the number of strengthenings.

6.6.2 *C11Tester Benchmarks (RQ1)*

We identified and fixed several (x86, Arm)-robustness errors in the C11Tester benchmark programs. We ran each program 1,000 times, or fewer if that identified the errors. Given a set of erroneous traces, we computed the cheapest fix and manually strengthened the accesses in the corresponding program. We repeated these steps until no more errors were found, upon which we reported the number of strengthenings and average final trace lengths in Figure 29.

Observe that only *three* of the programs needed robustness enforcement, the others Origami reported as (x86, Arm)-robust (for our adapted Arm model from Section 6.2.2). In spsc-queue, we found po-chains between the events generated from the W_{L} and $\texttt{RMW}_{\texttt{rlx}}$ accesses. Its unordered accesses are fixed in two steps $\texttt{RMW}_{\texttt{rlx}} \rightsquigarrow \texttt{RMW}_{\texttt{L}} \rightsquigarrow \texttt{RMW}_{\texttt{AL}}$. For the first violation, we identify the resulting pair of write events ordered in x86, but not in Arm. To fix the violation, our algorithm transforms $\texttt{RMW}_{\texttt{rlx}} \rightsquigarrow \texttt{RMW}_{\texttt{L}}$. In the next step, we identify the unordered po-edge from $\texttt{rlx}_{\texttt{L}}$ to $\mathsf{R}_{\texttt{rlx}}$ events generated from the same accesses. These are ordered in x86 but not in Arm. Hence, we perform the transformation $\texttt{RMW}_{\texttt{L}} \rightsquigarrow \texttt{RMW}_{\texttt{AL}}$. We already showed a violation in mpmc-queue in Figure 26 and its transformed version. We found several errors in ms-queue-tsan11, one of which Figure 30 illustrates.

Finally, we observed Origami often suggests similar strengthenings for the same program, but because of its random nature may suggest a different fix with equal cost. Consequently, depending on the order, we may obtain an altogether different set of fixes for the same program, both independently fully fixing it. Unfortunately, sometimes this process results in *redundant* strengthenings, which is a strengthening that is no longer required in the fully fixed program. This is a consequence of the random nature of Origami where it ends up at a *local* minimum.

6.6.3 *Comparison with Fency (RQ2)*

Figure 31 compares our Origami against Fency (Chakraborty, 2021) on the Fency benchmarks. Fency inserts fences after *static* analysis, thus overapproximating the necessary fence locations. Note that Fency inserts *fences* between events (*i.e.,* F, $\mathsf{F}_{\textsc{ld}}$, $\mathsf{F}_{\textsc{st}}$), whereas we *strengthen accesses* with acquire and/or release. We *modified* Fency's original benchmarks because the original variants could not run; Fency's *static* analysis

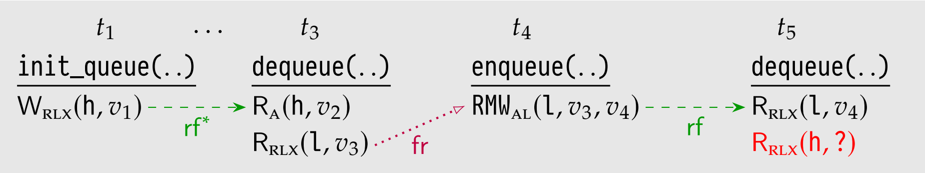

t_1
...
t_3
t_4
t_5
init_queue(..)
dequeue(..)
enqueue(..)
dequeue(..)
$\mathrm{W_{RLX}}(\mathtt{h}, v_1)$
rf*
$\mathrm{R_A}(\mathtt{h}, v_2)$
$\mathrm{R_{RLX}}(\mathtt{l}, v_3)$
fr
$\mathrm{RMW_{AL}}(\mathtt{l}, v_3, v_4)$
rf
$\mathrm{R_{RLX}}(\mathtt{l}, v_4)$
$\mathrm{R_{RLX}}(\mathtt{h}, ?)$

Figure 30: Violation in ms-queue-tsan11 (simplified), spanning *five* threads. The final read event on h in t_5 is ordered after the write in t_1 on x86, but not on Arm. This violation only showed up in *two* of the 1,000 traces and was at least 84 events apart in those traces. (Thread 2 with an RMW on h omitted)

	trace	Origami		Fency		Risotto	
name	length	#	rt	#	rt	#	rt
barrier	19	0	–	0	–	4	–
barrier-ext	67	2	23	2	23	3	23
dekker	30	0	–	4	–	20	–
peterson	28	2	–	4	–	10	–
peterson-ext	251	3	38	6	49	5	48
lamport	184	3	63	8	63	13	63
spinlock	39	0	31	0	31	3	31
ticketlock	57	0	32	1	32	2	32
seqlock	233	6	31	8	32	8	32
rcu-offline	1,283	8	13	12	29	19	28
cilk	1,879	3	70	8	98	17	129

Figure 31: Fency Benchmarks. Trace lengths averaged over 1,000 runs. # is the number of strengthenings, and 'rt' is the runtime in ms. The missing runtimes were dominated by process spawning overhead.

never needed executable programs. However, we need to execute programs to obtain traces. We also executed Fency on these modified programs. To ensure a fair runtime comparison, we compiled all variants of these programs with Fency's Clang version (~10.0.0).

Figure 31 shows the comparison with our Origami. Observe that Fency strengthens *at least as much* as our Origami. Fency reports robustness violations in dekker and ticketlock benchmarks, which are false positives as it *overapproximates* memory aliasing with its static analysis. In contrast, our approach analyzes concrete traces and identifies these benchmarks as robust. For the other benchmarks, Fency inserts equal or more fences than our approach. Moreover, observe that programs obtained with Origami run *at least as fast* as those produced by Fency. The programs whose runtime is missing include only a few accesses across two threads; Their process spawn

	trace	Origami		Risotto	
name	length	#	runtime	#	runtime
`histogram`	8,778	0	134	18	193
`linear_regression`	7,786	0	128	22	151
`pca`	3,233	0	847	38	931
`string_match`	◊6,191	0	295	25	2,349
`kmeans`	9,225	0	410	53	670
`matrix_multiply`	2,486	0	2,442	32	2,740
`word_count` (partial)	◊10,524	0	386	80	896

Figure 32: Phoenix Benchmarks. Most trace lengths are averaged over 1,000 runs, but ◊ marks smaller sample sizes of 100 for more expensive programs. # is the number of strengthenings, and the runtime is given in ms.

cost eclipses their runtime and thus doesn't give any meaningful result. Finally, we observe that Origami often produces and analyzes traces within *seconds*; However, when scaling this up *significantly*, we observe a dynamic analysis time of 31 minutes *per trace*. So, to answer **RQ2**, we produce fewer (or equal) strengthenings to Fency and have better (or equal) runtimes of the generated Arm programs.

6.6.4 *Lasagne/Risotto Comparison: Phoenix (RQ3)*

Finally, we compare against Risotto's mapping scheme. Lasagne and Risotto are *binary translators*, which translate *binary* x86 programs to Armv8. That translation incurs overhead (~51% for Lasagne and much more for Risotto) compared to the native LLVM compilation to Arm, rendering direct comparison unfair. Instead, we manually apply its *mapping scheme* to the Phoenix (Ranger et al., 2007) programs, thus avoiding the lifting overhead. We compare the number of strengthenings and program runtimes (compiled with LLVM 18.1.8), shown in Figure 32. For Origami's dynamic analysis, we reduced inputs significantly (*e.g.*, reducing `histogram`'s image from ~35M pixels to 500), but used the original large inputs to measure the final program runtimes. Besides the Phoenix benchmarks, we also applied Risotto's scheme to the Fency benchmarks in Figure 31. As Fency does not support various features present in the Phoenix benchmarks, we could not analyze them with Fency.

Interestingly, Origami indicates none of the Phoenix programs require strengthening. The threads *don't* concurrently modify the same variables, implying that all Phoenix programs are embarrassingly parallel. For the Phoenix programs, Risotto's scheme requires more fences than Origami and produces slower (or equal) programs. Observe that `string_match` and `word_count` particularly slow down (8× and 2.3× respectively) due to fences inserted in frequently called standard library functions, such as `strlen` and `strcmp`. We observe similar results on the Fency benchmarks in Figure 31, where Risotto's scheme inserts more fences and is slower than (or equal to) both Fency and ORIGAMI. However, note that RISOTTO's scheme is *correct in general* and thus works without program analysis. We observe that ORIGAMI runs particularly slowly on `histogram`, as it has $3 \cdot 256$ memory locations *per thread*, observed by all subsequent events.

6.7 RELATED WORK

State-of-the-art compilers such as LLVM (LLVM Team, 2025; Lattner and Adve, 2004), GCC, and binary translators follow respective mapping schemes Sewell (2011). These mapping schemes ensure correct translation for all contexts. However, as our experiments noted, in many scenarios, these schemes introduce additional fences or stronger memory accesses, which are unnecessary and detrimental to performance.

Various approaches identify the differences in behaviors for a given program based on static analysis (Alglave and Maranget, 2011; Chakraborty, 2021; Alglave et al., 2017; Beck et al., 2023), model checking (Lahav and Margalit, 2019; Lahav and Boker, 2022; Oberhauser et al., 2021; Bouajjani et al., 2013a,b), and solvers (Ponce de León et al., 2017; Wickerson et al., 2017). The static analysis approaches derive abstract graphs from a program for analyzing possible executions (Alglave and Maranget, 2011; Alglave et al., 2017; Chakraborty, 2021; Beck et al., 2023). Musketeer by Alglave and Maranget (2011); Alglave et al. (2017) checks if the program has any non-SC execution. Fency by Chakraborty (2021) also check the differences between weak models including x86 and Arm. However, these tools overapproximate the program behaviors to perform sound analyses and as a tradeoff, suffer from false positives as noted for Fency. AtoMig by Beck et al. (2023) adopted a static analysis yet unsound approach to port programs written for x86 to Arm, and thus also lacks completeness. We could not compare with AtoMig as it is not publicly available.

There are model-checking-based approaches that check SC-robustness under weak memory models (Lahav and Margalit, 2019; Lahav and Boker, 2022; Oberhauser et al., 2021; Bouajjani et al., 2013a,b). Unlike these, we check the differences between a pair of weak memory models. Moreover, while model checkers can precisely explore all executions, they are costly in usage and performance. PORTHS by Ponce de León et al. (2017) uses an SMT solver for bounded portability analysis from Power to TSO. Memalloy by Wickerson et al. (2017) uses the alloy solver to identify an execution that differentiates between memory models. However, it does not scale to large programs. Moreover, these existing approaches do not perform any required transformations.

Although several model checkers listed above build on an operational semantics, sometimes implicitly, other operational-axiomatic semantics were explicitly developed before. Particularly, Doherty et al. (2019) propose an operational semantics for RC11 (Lahav et al., 2017) that disallows $\mathsf{po} \cup \mathsf{rf}$-cycles, as we do. In contrast, another operational RC11 model by Wright et al. (2021) accurately allows those cycles.

6.8 CONCLUSION

We presented Origami, a robustness analysis and enforcement approach to port concurrent applications written for x86 to Arm. Our approach simulates the program's execution under both the x86 and Arm memory models and detects whether the latter shows additional behaviors. If so, Origami assists programmers by suggesting access strengthenings that eliminate those additional behaviors. We evaluated Origami on several benchmarks and compare against state-of-the-art approaches – the Fency static analyzer and the mapping schemes from Chapters 2 to 4. We showed that our approach inserts fewer fences than those existing approaches, producing programs that are 1.8× faster, on average.

Chapter 7

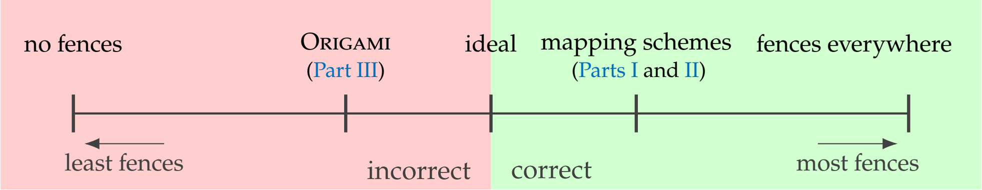
Conclusion

We conclude this dissertation by answering our research questions, for which we revisit the primary contributions of the chapters. We also discuss several key observations and future work, but start by repeating our research question:

Research Question

How can we correctly translate programs between weak memory model architectures while minimizing performance overhead?

To answer this question, we explored two mechanisms to translate programs between weak memory model architectures. In Part I we proposed mapping schemes, which we proved correct. The corresponding proof challenges and techniques we explained in Part II. In Part III we proposed an alternative algorithm to insert fences after detecting violations while simulating the program under its semantics. We can visualize those solutions as follows:



The "ideal" solution is a hypothetical perfect solution ordering *exactly* those instructions needed in any possible execution of the program. Unfortunately, obtaining such a solution is impossible (Rice, 1953), meaning an algorithm producing that solution *cannot exist*. On the left extreme, we insert no fences at all, which is clearly incorrect. On the right extreme, we insert full fences between any two adjacent memory instructions, which is clearly correct but significantly harms performance.

In-between those extremes, we have our own solutions, improving upon the extremes and approaching the ideal from both sides. Our mapping schemes (Part I) are *correct in general*, for any program with any number of threads. These schemes from x86 to Arm and RISC-V place significantly fewer fences than the naive approach, benefiting performance. However, these schemes cannot consider the exact set of orders needed at runtime, for instance, because of specific interactions between threads. Hence, the mapping schemes *over-approximate* the ideal solution, sometimes inserting too many fences. In contrast, our dynamic analysis Origami (Part III) exactly orders those instructions needed, for any execution explored during random analysis. That analysis *under-approximates* the ideal, as it will fail to explore many of the infinitely-many executions. However, it inserts significantly fewer fences than the mapping schemes. Neither of our mechanisms can be ideal but approach it from both sides. We elaborate on those approaches while addressing the sub-questions, highlighting their respective key observations.

Sub-Question 1

How can we define performant *mapping schemes* between weak memory model architectures and prove these preserve robustness?

Mapping Schemes. In Part I we defined formal mapping schemes between memory accesses from x86 to Arm and RISC-V. Besides proving them correct, we demonstrate their usability in three binary translators. Although the resulting mappings between architectures were similar, they supported different architectural features and translated through various intermediate representations, specific to each translator.

In Chapter 2, we defined mapping schemes for Lasagne, our static binary translator, which internally uses LLVM IR. After identifying errors in the existing translation of LLVM and lifter mctoll, we defined our own intermediate memory model LIMM, through which Lasagne's mapping schemes translated from x86 to Arm. LIMM supports various transformations that existing LLVM optimizations rely on, such as instruction reordering and fence elimination, which we also proved correct.

In Chapter 3, we addressed a similar challenge for our dynamic binary translator Risotto that builds upon QEMU. There too, we identified robustness errors in QEMU's existing translation, which we fixed with new mappings over our own TIMM model. The primary challenge was supporting Arm's new $\texttt{CAS}_{\texttt{AL}}$ instruction (*i.e.,* single-instruction RMW), whose semantics were only described in a newer Arm model (Alglave et al., 2021). Targeting those new semantics required significant reworking of Lasagne's proofs to target the new Arm model for Risotto. Through these extensive proof efforts, we identified an issue in those semantics as $\texttt{CAS}_{\texttt{AL}}$ ordered weaker than it should, which we fixed in the Arm model. This discovery particularly highlights the value of proof mechanization.

Finally, in Chapter 4 we again faced a similar challenge for our hybrid binary translator Arancini. Arancini was built from the ground up, meaning we could control its semantics and implementation. We again defined an intermediate model AIMM, resembling the prior models. That model and its proofs addressed two new primary challenges: *(i)* supporting mixed-size accesses and *(ii)* targeting RISC-V. The first we address by including mixed-size semantics in our AIMM model and proving a corresponding mapping correct from x86 to Arm through AIMM, with mixed-size accesses. Those proofs also required significantly overhauling the existing proof structure to support those mixed-size models. Finally, the mapping to RISC-V required its own separate mechanized proofs.

Proof Challenges. Through the challenges we faced while mechanizing the numerous proofs for the mappings in Part I, we discovered various common proof structures, which we discussed in Chapter 5 (in Part II). In particular, we discovered that most mappings preserve well-formedness—consisting of 27 properties—in a similar way. Repeatedly proving those properties is very time-consuming. To address that, we identified several simpler properties and relations which imply that a mapping preserves well-formedness – we can thus prove it generally, saving much proof engineering efforts.

Additionally, we identified common mechanisms with which the proofs mapped properties and relations between execution graphs. That allowed us to define abstractions with corresponding primitives, capturing that proof structure. For instance, primitive relations (*i.e.*, po, rf, mo, and rmw) often map in similar ways; while derived relations are often composed in similar ways (*e.g.*, with ';', '∪', or '∩'), which our mapping primitives capture. While we use those primitives in our general well-formedness proof, their primary value is their role in simplifying the mechanization of the mapping-specific proof fragments (*e.g.*, consistency). We have implemented these general proofs and mapping primitives in our Agda proof library Burrow.

Summary. We have defined various practical mapping schemes that correctly translate weak memory primitives from x86 to Arm and RISC-V, supporting various architectural features. We demonstrated their usability by implementing them in binary translators, while proving their correctness with mechanized proofs. Finally, we developed a general proof framework to simplify proofs mechanization for future mapping schemes.

Sub-Question 2

How can we detect and fix robustness violations that appear in program traces?

In Part III we approach the x86-Arm robustness challenge from the other side. Although our mapping schemes consider an *over-approximation* of the program's behavior to ensure robustness, they often insert too many fences. In particular, they must be correct in general, thus assuming unknown threads communicate arbitrarily; We had to consider all possible behaviors when defining the mapping schemes. In contrast, when we know the specific program, we can simulate it under the formal semantics to detect and fix program-specific violations, which Origami did in Chapter 6. Origami analyses the runtime behavior of programs by simulating them step-by-step under both the x86 and Arm weak memory models to detect x86-Arm robustness violations. It thus *under-approximates* the set of behaviors; Though it often correctly fixes violations, its randomized analysis could miss violations. We demonstrate programs fixed using Origami are on average 1.8× faster than those using the mapping schemes, across the benchmarks we used.

7.1 FUTURE WORK

Approach Ideal Solution. We have translated programs between weak memory architectures in various ways. Our mapping schemes are correct in the general case, but necessarily *over-approximate* a program's semantics, sometimes placing too many fences. Our dynamic analysis technique only places those fences needed to fix violations detected at runtime, *under-approximating* a program's semantics and thus sometimes placing too few fences. The crucial observation here is that they approach the hypothetical ideal solutions (which is undecidable) from either side. We foresee opportunities to combine aspects of both to get closer to the ideal solution, placing fewer fences than our mapping schemes but remaining correct in general. Instead of over-approximating the behavior of any program, we could over-approximate the behavior of a specific program. For instance, a program analysis could over-approximate the communication between threads, which mapping schemes then fix.

Intermediate Weak Memory Model. In Part I we introduced the three memory models LIMM, TIMM, and AIMM, which were intermediate within our mapping schemes. Our model preserves the order explicitly among memory accesses with fences, but *does not* order them with *dependencies* (*e.g.,* '`a = X; Y = a;`'), unlike Arm (Pulte et al., 2017; Alglave et al., 2021). We decided to eliminate dependencies to work with existing translation system –LLVM for LIMM and TCG for TIMM– whose existing optimizations would likely not respect those dependencies. Although that decision poses no issues for our context, when translating from x86 to Arm and RISC-V, it limits extension to others; In particular, any mapping from Arm to itself through any of our models *cannot be* the identity mapping. The dependencies of the original Arm program are not preserved along any mapping –because our models do not represent them– meaning an additional fence must be inserted to represent those orders explicitly. When mapping back to Arm, possibly after optimizing the intermediate program, those fences remain, resulting in a program with more fences. To avoid those unnecessary fences, we need another intermediate model that captures the intricate ordering constraints of weaker architectures.

Detect po∪rf cycles. Origami (Chapter 6) detects x86-Arm robustness violations by simulating a program step-by-step. When simulating a load instruction, which generates a R event, it reads a value from an existing W event, generating a rf edge between them. The algorithm thus always executes a store before the load observing it. However, as Arm can execute R−W pairs out-of-order, that assumption does not always hold. Specifically, Origami cannot detect '$\mathsf{po} \cup \mathsf{rf}$' cycles (Lahav et al., 2017; Geeson and Smith, 2024; Lee et al., 2023), which is why we strengthened the Arm model it uses (Section 6.2.2). It remains future work to adapt its algorithm to detect those cycles.

Acronyms

CAS	Compare-and-Swap
DBT	Dynamic Binary Translation
HBT	Hybrid Binary Translation
IP	Instruction Pointer
IR	Intermediate Representation
ISA	Instruction Set Architecture
LEM	Law of the Excluded Middle
ll/sc	load-linked store-conditional
LTS	Labeled Transition System
RAR	Read-After-Read
RAW	Read-After-Write
RMW	Read-Modify-Write
SBT	Static Binary Translation
SC	Sequential Consistency
TCG	Tiny Code Generator
TSO	Total Store Order
WAW	Write-After-Write

Bibliography

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. 2015b. Precise and Sound Automatic Fence Insertion Procedure under PSO. In *NETYS (Lecture Notes in Computer Science, Vol. 9466)*. 32–47. https://doi.org/10.1007/978-3-319-26850-7_3

Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. 2015a. The Best of Both Worlds: Trading Efficiency and Optimality in Fence Insertion for TSO. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 308–332. https://doi.org/10.1007/978-3-662-46669-8_13

Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *IEEE Computer* 29, 12 (1996), 66–76. https://doi.org/10.1109/2.546611

Sarita V. Adve and Mark D. Hill. 1993. A Unified Formalization of Four Shared-Memory Models. *IEEE Trans. Parallel Distrib. Syst.* 4, 6 (June 1993), 613 – 624. https://doi.org/10.1109/71.242161

Agda Team. 2025a. Agda Documentation. https://agda.readthedocs.io/en/stable/.

Agda Team. 2025b. Agda standard library. https://github.com/agda/agda-stdlib.

Agda Team. 2025c. *PapersUsingAgda - The Agda Wiki*. https://wiki.portal.chalmers.se/agda/Main/PapersUsingAgda

Jade Alglave. 2012. A Formal Hierarchy of Weak Memory Models. *Form. Methods Syst. Des.* 41, 2 (2012), 178 – 210. https://doi.org/10.1007/s10703-012-0161-5

Jade Alglave. 2022. *GitHub issue: "[AArch64 cat] Atomics strengthening"*. https://github.com/herd/herdtools7/pull/322

Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 8 (July 2021), 54 pages. https://doi.org/10.1145/3458926

Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2017. Don' t Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion. 39, 2, Article 6 (May 2017), 38 pages. https://doi.org/10.1145/2994593

Jade Alglave and Luc Maranget. 2011. Stability in Weak Memory Models. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 50–66. https://doi.org/10.1007/978-3-642-22110-1_6

Jade Alglave and Luc Maranget. 2025. `aarch64.cat` – *Armv8* `cat` *model*. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat Accessed on 2025-02-05.

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Computer Aided Verification*. 258–272. https://doi.org/10.1007/978-3-642-14295-6_25

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 40. https://doi.org/10.1145/2594291.2594347

AMD 2024. AMD64 Architecture Programmer ' s Manual, Volume 2, System Programming.

Kristy Andrews and Duane Sand. 1992. Migrating a CISC computer family onto RISC via object code translation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, USA) *(ASPLOS V)*. Association for Computing Machinery, New York, NY, USA, 213 – 222. https://doi.org/10.1145/143365.143520

Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16)*. USENIX Association, USA, 583 – 600.

Apple Inc. 2021. Rosetta 2 on a Mac with Apple silicon. https://support.apple.com/fr-fr/guide/security/secebb113be1/web.

Apple Inc. 2023. *Apple unveils M3, M3 Pro, and M3 Max, the most advanced chips for a personal computer*. https://www.apple.com/newsroom/2023/10/apple-unveils-m3-m3-pro-and-m3-max-the-most-advanced-chips-for-a-personal-computer/

Arm. 2016. ARM Cortex-A72 MPCore Processor Technical Reference Manual – Memory access sequence. https://developer.arm.com/documentation/100095/0003/Memory-Management-Unit/Memory-access-sequence.

Arm Limited 2024. *Arm® Architecture Reference Manual*. Arm Limited. https://developer.arm.com/documentation/ddi0487/latest/.

Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2012. What's Decidable about Weak Memory Models?. In *ESOP'12*. 26–46. https://doi.org/10.1007/978-3-642-28869-2_2

Avast Developers. 2022. A retargetable machine-code decompiler based on LLVM. https://github.com/avast/retdec.

Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 634 – 648. https://doi.org/10.1145/2837614.2837637

Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. *SIGPLAN Not.* 47, 1 (Jan. 2012), 509 – 520. https://doi.org/10.1145/2103621.2103717

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 55 – 66. https://doi.org/10.1145/1926385.1926394

Martin Beck, Koustubha Bhat, Lazar Stricevic, Geng Chen, Diogo Behrens, Ming Fu, Viktor Vafeiadis, Haibo Chen, and Hermann Härtig. 2023. AtoMig: Automatically Migrating Millions Lines of Code from TSO to WMM. In *ASPLOS 2023*. 61–73. https://doi.org/10.1145/3575693.3579849

Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

Yves Bertot and Pierre Castran. 2010. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions* (1st ed.). Springer Publishing Company, Incorporated.

Christian Bienia. 2011. *Benchmarking modern multiprocessors*. Ph. D. Dissertation. Princeton University, USA. Advisor(s) Kai Li. AAI3445564.

Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013a. Checking and Enforcing Robustness against TSO. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 533–553.

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2013b. Verifying concurrent programs against sequential specifications. In *ESOP'13*. Springer, 290–309.

Ahmed Bougacha. 2022. Binary Translator to LLVM IR. https://github.com/repzret/dagger.

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda – A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics*. 73–78.

Soham Chakraborty. 2021. Robustness between Weak Memory Models. In *2021 Formal Methods in Computer Aided Design (FMCAD)*. 173–182. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_26

Soham Chakraborty and Viktor Vafeiadis. 2016. Validating optimizations of concurrent C/C++ programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 216 – 226. https://doi.org/10.1145/2854038.2854051

Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the concurrency semantics of an LLVM fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 100 – 110.

Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding Thin-Air Reads with Event Structures. In *POPL*, Vol. 3. https://doi.org/10.1145/3290383

Jesper Cockx and Andreas Abel. 2018. Elaborating dependent (co)pattern matching. *Proc. ACM Program. Lang.* 2, ICFP, Article 75 (July 2018), 30 pages. https://doi.org/10.1145/3236770

Thierry Coquand. 1992. Pattern Matching with Dependent Types.

Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. 2017. Cross-ISA Machine Emulation for Multicores. In *CGO'2017*. IEEE Press, 210 – 220. https://doi.org/10.1109/CGO.2017.7863741

Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. 2011. PQEMU: A Parallel System Emulator Based on QEMU. In *ICPADS'11*. IEEE, Tainan, Taiwan, China, 276–283. https://doi.org/10.1109/ICPADS.2011.102

Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2019. Verifying C11 programs operationally. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 355 – 365. https://doi.org/10.1145/3293883.3295702

Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 413–430.

Reinoud Elhorst. 2014. Lowering C11 Atomics for ARM in LLVM. In *European LLVM Conference*.

Robert W. Floyd. 1993. *Assigning Meanings to Programs*. Springer Netherlands, Dordrecht, 65–81. https://doi.org/10.1007/978-94-011-1793-7_4

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 *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 429 – 442. https://doi.org/10.1145/3009837.3009839

Sheng-Yu Fu, Ding-Yong Hong, Yu-Ping Liu, Jan-Jan Wu, and Wei-Chung Hsu. 2018. Efficient and retargetable SIMD translation in a dynamic binary translator. *Software: Practice and Experience* 48, 6 (2018), 1312–1330. https://doi.org/10.1002/spe.2573 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2573

Natalia Gavrilenko, Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 355–365.

Luke Geeson and Lee Smith. 2024. Compiler Testing with Relaxed Memory Models. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 334–348. https://doi.org/doi.org/10.1109/CGO57630.2024.10444836 preprint: https://lukegeeson.com/assets/publications/cgo24/paper.pdf.

Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, USA) *(ISCA '90)*. Association for Computing Machinery, New York, NY, USA, 15 – 26. https://doi.org/10.1145/325164.325102

Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and Andrew W. Roscoe. 2014. FDR3 — A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 187–201.

Redha Gouicem, Dennis Sprokholt, Jasper Ruehl, Rodrigo C. O. Rocha, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 107 – 122. https://doi.org/10.1145/3567955.3567962

Yu-Chuan Guo, Wuu Yang, Jiunn-Yeu Chen, and Jenq-Kuen Lee. 2016. Translating the ARM Neon and VFP instructions in a binary translator. *Software: Practice and Experience* 46, 12 (2016), 1591–1615. https://doi.org/10.1002/spe.2394 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2394

Lisa Higham, Lillanne Jackson, and Jalal Kawash. 2007. Specifying Memory Consistency of Write Buffer Multiprocessors. *ACM Trans. Comput. Syst.* (2007). https://doi.org/10.1145/1189736.1189737

Lisa Higham, Jalal Kawash, and Nathaly Verwaal. 1997. Defining and Comparing Memory Consistency Models. In *PDCS'97*.

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576 – 580. https://doi.org/10.1145/363235.363259

Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. 2012. HQEMU: A Multi-Threaded and Retargetable Dynamic Binary Translator on Multicores. In *CGO'12*. 104 – 113. https://doi.org/10.1145/2259016.2259030

Intel Corporation 2025. *Intel® 64 and IA-32 Architectures Software Developer Manual*. Intel Corporation.

Saagar Jha. 2020. TSOEnabler – Kernel extension that enables TSO for Apple silicon processes. https://github.com/saagarjha/TSOEnabler.

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 175 – 189. https://doi.org/10.1145/3009837.3009850

Hyungseok Kim, Soomin Kim, and Sang Kil Cha. 2025. Towards Sound Reassembly of Modern x86-64 Binaries. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1317 – 1333. https://doi.org/10.1145/3676641.3716026

Arun Kishan and Rani Borkar. 2024. *Azure Cobalt 100-based Virtual Machines are now generally available*. https://azure.microsoft.com/en-us/blog/azure-cobalt-100-based-virtual-machines-are-now-generally-available/

Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. 2023. Kater: Automating Weak Memory Model Metatheory and Consistency Checking. *Proc. ACM Program. Lang.* 7, POPL, Article 19 (Jan. 2023), 29 pages. https://doi.org/10.1145/3571212

Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A Model Checker for Weak Memory Models. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20 – 23, 2021, Proceedings, Part I*. Springer-Verlag, Berlin, Heidelberg, 427 – 440. https://doi.org/10.1007/978-3-030-81685-8_20

Ori Lahav and Udi Boker. 2022. What' s Decidable About Causally Consistent Shared Memory? *ACM Trans. Program. Lang. Syst.* 44, 2, Article 8 (apr 2022), 55 pages. https://doi.org/10.1145/3505273

Ori Lahav and Roy Margalit. 2019. Robustness against Release/Acquire Semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 126 – 141. https://doi.org/10.1145/3314221.3314604

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 618 – 632. https://doi.org/10.1145/3062341.3062352

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Leslie Lamport. 2019. *Time, clocks, and the ordering of events in a distributed system*. Association for Computing Machinery, New York, NY, USA, 179 – 196. https://doi.org/10.1145/3335772.3335934

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. San Jose, CA, USA, 75–88. https://doi.org/10.1109/CGO.2004.1281665

Jaejin Lee and David A. Padua. 2001. Hiding Relaxed Memory Consistency with Compilers. *IEEE Trans. Comput.* 50, 8 (2001), 824–833. https://doi.org/10.1109/PACT.2000.888336

Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, and Ori Lahav. 2023. Putting Weak Memory in Order via a Promising Intermediate Representation. *Proc. ACM Program. Lang.* 7, PLDI (2023). https://doi.org/10.1145/3591297

Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 362 – 376. https://doi.org/10.1145/3385412.3386010

Alexander Linden and Pierre Wolper. 2011. A Verification-Based Approach to Memory Fence Insertion in Relaxed Memory Systems. In *SPIN'11*. 144–160.

Alexander Linden and Pierre Wolper. 2013. A Verification-Based Approach to Memory Fence Insertion in PSO Memory Systems. In *TACAS*.

Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. 2012. Dynamic Synthesis for Relaxed Memory Models. In *PLDI '12*. 429–440. https://doi.org/10.1145/2345156.2254115

Nian Liu, Binyu Zang, and Haibo Chen. 2020. No Barrier in the Road: A Comprehensive Study and Optimization of ARM Barriers. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 348 – 361. https://doi.org/10.1145/3332466.3374535

LLVM Team. 2025. The LLVM Compiler Infrastructure. https://llvm.org/.

Weiyu Luo and Brian Demsky. 2021. C11Tester: A Race Detector for C/C++ Atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 630 – 646. https://doi.org/10.1145/3445814.3446711

Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. 2015. ArMOR: Defending against Memory Consistency Model Mismatches in Heterogeneous Architectures. In *ISCA'15*. 388 – 400. https://doi.org/10.1145/2749469.2750378

Sela Mador-Haim, Rajeev Alur, and Milo M K. Martin. 2010. Generating Litmus Tests for Contrasting Memory Consistency Models. In *CAV'10*. 273 – 287. https://doi.org/10.1007/978-3-642-14295-6_26

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.

Roy Margalit and Ori Lahav. 2021. Verifying Observational Robustness against a C11-Style Memory Model. *Proc. ACM Program. Lang.* 5, POPL, Article 4 (2021). https://doi.org/10.1145/3434285

Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. 2025. Model Checking C/C++ with Mixed-Size Accesses. *Proc. ACM Program. Lang.* 9, POPL, Article 75 (Jan. 2025), 21 pages. https://doi.org/10.1145/3704911

Per Martin-Löf. 1982. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science VI*. Studies in Logic and the Foundations of Mathematics, Vol. 104. Elsevier, 153–175. https://doi.org/10.1016/S0049-237X(09)70189-2

Conor McBride. 2012. A polynomial testing principle. https://personal.cis.strath.ac.uk/conor.mcbride/PolyTest.pdf (2012).

Microsoft. 2024. *Windows on ARM - How emulation works on Arm*. https://learn.microsoft.com/en-us/windows/arm/apps-on-arm-x86-emulation#prism

Robin Morisset and Francesco Zappa Nardelli. 2017. Partially redundant fence elimination for x86, ARM, and power processors. In *CC'17*. 1–10. https://doi.org/10.1145/3033019.3033021

Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI'13*. ACM, 187–196. https://doi.org/10.1145/2499370.2491967

Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.

Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 530 – 545. https://doi.org/10.1145/3445814.3446748

Scott Owens. 2010. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *Proceedings of the 24th European Conference on Object-Oriented Programming* (Maribor, Slovenia) *(ECOOP'10)*. Springer-Verlag, Berlin, Heidelberg, 478 – 503.

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Gustavo Petri, Jan Vitek, and Suresh Jagannathan. 2015. Cooking the Books: Formalizing JMM Implementation Recipes. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, Vol. 37. 445–469. https://doi.org/10.4230/LIPIcs.ECOOP.2015.445

Gordon D. Plotkin. 1981. A structural approach to operational semantics. (1981).

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 (2019). https://doi.org/10.1145/3290382

Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2017. Portability Analysis for Weak Memory Models PORTHOS: One Tool for all Models. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 299–320.

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158107

QEMU Team. 2003. QEMU: A generic and open source machine emulator and virtualizer. https://www.qemu.org/.

QEMU Team. 2021. QEMU Wiki: Features/tcg-multithread. https://wiki.qemu.org/Features/tcg-multithread.

QEMU Team. 2025. Atomic operations in QEMU. https://www.qemu.org/docs/master/devel/atomics.html.

Azalea Raad, Ori Lahav, John Wickerson, Piotr Balcer, and Brijesh Dongol. 2024. Intel PMDK Transactions: Specification, Validation and Concurrency. In *Programming Languages and Systems*, Stephanie Weirich (Ed.). Springer Nature Switzerland, Cham, 150–179.

Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA*. IEEE Computer Society, Scottsdale, AZ, USA, 13–24.

Henry G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. http://www.jstor.org/stable/1990888

RISC-V International 2024. *The RISC-V Instruction Set Manual: Volume I*. RISC-V International. https://riscv.org/specifications/ratified/.

Rodrigo C. O. Rocha, Dennis Sprokholt, Martin Fink, Redha Gouicem, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. Lasagne: A Static Binary Translator for Weak Memory Model Architectures. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 888 – 902. https://doi.org/10.1145/3519939.3523719

Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 311 – 322. https://doi.org/10.1145/2254064.2254102

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 175 – 186. https://doi.org/10.1145/1993498.1993520

Shigeyuki Sato, Taiyo Mizuhashi, Genki Kimura, and Kenjiro Taura. 2025. Efficiently Adapting Stateless Model Checking for C11/C++11 to Mixed-Size Accesses. In *Programming Languages and Systems*, Oleg Kiselyov (Ed.). Springer Nature Singapore, Singapore, 346–364.

Jaroslav Ševčík. 2011. Safe optimisations for shared-memory concurrent programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 306 – 316. https://doi.org/10.1145/1993498.1993534

Jaroslav Ševčík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP 2008 – Object-Oriented Programming*, Jan Vitek (Ed.). 27–51.

Peter Sewell. 2011. C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89 – 97. https://doi.org/10.1145/1785414.1785443

Dennis E. Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (1988), 282–312. https://doi.org/10.1145/42190.42277

Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. 2012a. LLBT: An LLVM-Based Static Binary Translator. In *CASES 2012*. 51 – 60. https://doi.org/10.1145/2380403.2380419

Bor-Yeh Shen, Jyun-Yan You, Wuu Yang, and Wei-Chung Hsu. 2012b. An LLVM-based hybrid binary translation system. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. 229–236. https://doi.org/10.1109/SIES.2012.6356589

Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. 1993. Binary translation. *Commun. ACM* 36, 2 (Feb. 1993), 69 – 81. https://doi.org/10.1145/151220.151227

Tom Spink, Harry Wagstaff, and Björn Franke. 2019. A Retargetable System-Level DBT Hypervisor. In *USENIX Annual Technical Conference*. USENIX Association, 505–520. https://doi.org/10.1145/3302516.3307357

Robert C. Steinke and Gary J. Nutt. 2004. A unified theory of shared memory consistency. *J. ACM* 51, 5 (2004), 800–849. https://doi.org/10.1145/1017460.1017464

Sélbastien Stormacq. 2022. *New – Amazon EC2 C7g Instances, Powered by AWS Graviton3 Processors*. https://aws.amazon.com/blogs/aws/new-amazon-ec2-c7g-instances-powered-by-aws-graviton3-processors/

Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. 2005. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *PPOPP'05*. 2 – 13. https://doi.org/10.1145/1065944.1065947

Trail of Bits. 2022. Framework for lifting x86, amd64, and aarch64 program binaries to LLVM bitcode. https://github.com/lifting-bits/mcsema.

Viktor Vafeiadis. 2018. Hahn: A Coq library. https://github.com/vafeiadis/hahn.

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'15*. ACM, 209–220. https://doi.org/10.1145/2676726.2676995

Viktor Vafeiadis and Francesco Zappa Nardelli. 2011. Verifying Fence Elimination Optimisations. In *SAS'11 (LNCS, Vol. 6887)*. Springer, 146–162. https://doi.org/10.1007/978-3-642-23702-7_14

Amin Vahdat. 2024. *Introducing Google Axion Processors, our new Arm-based CPUs*. https://cloud.google.com/blog/products/compute/introducing-googles-new-arm-based-cpu

Jun Wang, Jianmin Pang, Xiaonan Liu, Feng Yue, Jie Tan, and Liguo Fu. 2019. Dynamic Translation Optimization Method Based on Static Pre-Translation. *IEEE Access* 7 (2019), 21491–21501. https://doi.org/10.1109/ACCESS.2019.2897611

Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. 2011. COREMU: a scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. Association for Computing Machinery, New York, NY, USA, 213 – 222. https://doi.org/10.1145/1941553.1941583

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *POPL'17*. ACM, 190–204. https://doi.org/10.1145/3009837.3009838

Daniel Wright, Mark Batty, and Brijesh Dongol. 2021. Owicki-Gries Reasoning for C11 Programs with Relaxed Dependencies. In *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20 – 26, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 237 – 254. https://doi.org/10.1007/978-3-030-90870-6_13

S. Bharadwaj Yadavalli and Aaron Smith. 2019. Raising Binaries to LLVM IR with MCTOLL (WIP Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Phoenix, AZ, USA) *(LCTES 2019)*. Association for Computing Machinery, New York, NY, USA, 213 – 218. https://doi.org/10.1145/3316482.3326354

Zhaoxin Yang, Xuehai Chen, Liangpu Wang, Weiming Guo, Dongru Zhao, Chao Yang, and Fuxin Zhang. 2024. MFHBT: Hybrid Binary Translation System with Multi-stage Feedback Powered by LLVM. In *Advanced Parallel Processing Technologies*, Chao Li, Zhenhua Li, Li Shen, Fan Wu, and Xiaoli Gong (Eds.). Springer Nature Singapore, Singapore, 310–325.

Curriculum Vitae

Dennis Sprokholt earned his Computing Science bachelor degree cum laude from the University of Groningen. During his Computing Science master studies at Utrecht University, he specialized in Programming Technology and conducted research on superoptimization, also obtaining his master degree cum laude.

He started his PhD at Delft University of Technology, under the supervision of Soham Chakraborty and Eelco Visser, on the topic of binary translation; later under the supervision of Soham Chakraborty and Koen Langendoen. His PhD research contributed formal methods to translate programs from strong to weak memory model architectures. During his PhD, he co-supervised the research of many bachelor and master students. Dennis is currently a Humboldt postdoc at the Technical University of Munich, hosted by Pramod Bhatotia.

Titles in the IPA Dissertation Series since 2022

A. Fedotov. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical*. Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions*. Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition*. Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Extensions of (Concurrent) Kleene Algebra*. Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing*. Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving*. Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution*. Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems*. Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN*. Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification*. Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification*. Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software*. Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis*. Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More Effective Test Case Generation with Multiple Tribes of AI*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond*. Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic*. Faculty of Mathematics and Natural Sciences, UL. 2024-04

C.E. Brandt. *Test Amplification For and With Developers*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05

J.I. Hejderup. *Fine-Grained Analysis of Software Supply Chains*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06

J. Jacobs. *Guarantees by construction*. Faculty of Science, Mathematics and Computer Science, RU. 2024-07

O. Bunte. *Cracking OIL: A Formal Perspective on an Industrial DSL for Modelling Control Software*. Faculty of Mathematics and Computer Science, TU/e. 2024-08

R.J.A. Erkens. *Automaton-based Techniques for Optimized Term Rewriting*. Faculty of Mathematics and Computer Science, TU/e. 2024-09

J.J.M. Martens. *The Complexity of Bisimilarity by Partition Refinement*. Faculty of Mathematics and Computer Science, TU/e. 2024-10

L.J. Edixhoven. *Expressive Specification and Verification of Choreographies*. Faculty of Science, OU. 2024-11

J.W.N. Paulus. *On the Expressivity of Typed Concurrent Calculi*. Faculty of Science and Engineering, RUG. 2024-12

J. Denkers. *Domain-Specific Languages for Digital Printing Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-13

L.H. Applis. *Tool-Driven Quality Assurance for Functional Programming and Machine Learning*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-14

P. Karkhanis. *Driving the Future: Facilitating C-ITS Service Deployment for Connected and Smart Roadways*. Faculty of Mathematics and Computer Science, TU/e. 2024-15

N.W. Cassee. *Sentiment in Software Engineering*. Faculty of Mathematics and Computer Science, TU/e. 2024-16

H. van Antwerpen. *Declarative Name Binding for Type System Specifications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-01

I.N. Mulder. *Proof Automation for Fine-Grained Concurrent Separation Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2025-02

T.S. Badings. *Robust Verification of Stochastic Systems: Guarantees in the Presence of Uncertainty*. Faculty of Science, Mathematics and Computer Science, RU. 2025-03

A.M. Mir. *Machine Learning-assisted Software Analysis*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-04

L.T. Vinkhuijzen. *Data Structures for Quantum Circuit Verification and How To Compare Them*. Faculty of Mathematics and Natural Sciences, UL. 2025-05

D. van der Wal. *What is the Point? Single-Input-Change Testing a EULYNX Controller*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-06

A. Rosset. *Uniform Monad Presentations and Graph Quasitoposes*. Faculty of Sciences, Department of Computer Science, VU. 2025-07

L. Guo. *Higher-Order Termination with Logical Constraints*. Faculty of Science, Mathematics and Computer Science, RU. 2025-08

D.A. Manrique Negrin. *A Model Orchestra in Digital Twins: A Model-Driven Approach to Integration and Orchestration*. Faculty of Mathematics and Computer Science, TU/e. 2025-09

C.A. Esterhuyse. *Specification-Centric Multi-Agent Systems*. Faculty of Science, UvA. 2025-10

H.M. Muctadir. *Consistency Matters: Building Consistent Digital Twin Virtual Entities*. Faculty of Mathematics and Computer Science, TU/e. 2025-11

A. Stramaglia. *Model Checking Machine-Control Applications*. Faculty of Mathematics and Computer Science, TU/e. 2025-12

M. Saeedi Nikoo. *Supporting business process management: clone detection and recommendation techniques*. Faculty of Mathematics and Computer Science, TU/e. 2025-13

R.B. Rubbens. *Bridging the Implementation Gap: Advances in Model-Based Concurrent Program Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-14

F.I. van der Berg. *DMC Model Checker: Delta-Driven Variable-Length Next-State Generation via Recursive Compression*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-15

D.G. Sprokholt. *Correct Translation between Weak Memory Model Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-16