Specifying, Verifying, and Translating Between Memory Consistency Models

Daniel Joseph Lustig

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Electrical Engineering
Adviser: Professor Margaret Martonosi

November 2015

Abstract

Gone are the days when single-threaded performance was the primary metric of interest for a processor design. Driven by increasingly tight power constraints and by the slowing down of Moore's law and Dennard scaling, architects have turned to multicore parallelism and architectural heterogeneity as a means to continue delivering increases in chip performance and power efficiency. Both approaches can deliver improved performance per watt characteristics at the expense of creating systems which are dramatically more complex to design and harder to program.

In order to share data and coordinate their actions, the compute elements (e.g., CPU cores, GPU cores, accelerators) in modern processors communicate via *shared virtual memory*. In this model, regardless of the underlying physical reality, cores see the abstraction of a single unified address space shared with the other processing elements in the system. Communication through shared virtual memory takes place according to a *memory consistency model*—the set of rules and guarantees about the ordering and visibility of memory accesses issued by one processing element and observed by other processing elements. Unfortunately, in practice, hardware consistency model definitions frequently suffer from a lack of formalism, precision, and/or completeness, resulting in numerous situations in which the correct outcome(s) for a given scenario may be ambiguous or even simply undefined. To solve these problems, this thesis develops formal analysis techniques, specification formats, and automated tools that aim to mitigate the problems of incompleteness, imprecision, and/or incompatibility among memory consistency models.

First, this thesis proposes Memory Ordering Specification Tables (MOSTs), a systematic method for fully and explicitly enumerating the memory ordering requirements of axiomatic memory models. The architecture- and model-independent approach used by the MOST format allows for the direct comparison of preserved program order, fences, and other ordering enforcement mechanisms from the same

or even from different models. In particular, it allows for the direct comparison of models that are weakly-ordered (e.g., ARM, IBM Power, GPUs), strongly-ordered (e.g., sequential consistency, SPARC/x86 Total Store Ordering), or points in between. This thesis also presents the ArMOR framework for systematically analyzing and manipulating MOSTs. As a demonstration of the power of ArMOR, this thesis presents a methodology for automatically and dynamically translating code compiled under the assumptions of one memory model into code which can be executed by a (micro)architecture implementing a different memory model. By analyzing MOSTs for the source and destination architectures, ArMOR analysis can be used to produce optimized, self-contained translation engines called *shims*. The shim designs can be derived offline and in advance, and they can be easily implemented in software or hardware with manageable performance overhead.

Second, this thesis proposes PipeCheck, a framework for verifying the correctness of particular implementations of a given architectural memory model. PipeCheck defines a domain-specific language for specifying memory ordering behavior at the microarchitecture level. This language allows PipeCheck to explicitly specify and verify the behavior of common microarchitectural optimizations such as speculative load reordering which are intentionally abstracted away by higher-level models. PipeCheck's fast constraint solver software quantifies in just minutes whether an implementation is stronger than, weaker than, or equal in strength to its memory model requirements. It also reduces a currently intractable problem—verifying register transfer level (RTL) hardware descriptions against architectural memory model requirements—into a much more tractable task: verifying RTL against a set of locally-scoped, per-pipeline stage ordering properties. This thesis applies PipeCheck at a number of locations within a processor: within the pipeline, within the cache coherence protocol, and at the coherence/consistency interface, and it demonstrates the practicality of rigorously analyzing a broad range of microarchitectural scenarios.

In summary, this thesis presents architects with new techniques for reducing the complexity of defining memory consistency models and building systems that correctly implement them. In doing so, it narrows the existing gap between performance-first architectural design methodologies and rigorous verification of implementation correctness. As hardware and software complexity and dynamism continue to grow, the techniques developed in this thesis will help designers avoid the memory model bugs that continue to appear in hardware even today, and they will present programmers, compiler writers, and runtime systems with usable, precise descriptions of the memory ordering behaviors of any simple or complex computing system.

Acknowledgements

It's hard to believe that my time at Princeton is coming to an end. It was an amazing experience, and I can't help but acknowledge all of the help and friendship I received from so many people over my years here. None of this would have been possible without all of their support and guidance, and I'm going to do my best to thank them all here.

First of all, I would like to give enormous thanks to my adviser Margaret Martonosi. I couldn't have asked for anything more from an adviser. She helped guide me through the highs and the lows of my Ph.D., and she always pushed me to improve in all aspects of my career. She was always there for me whenever I needed advice, whether it be technical, professional, personal, or otherwise. She accepted nothing less than my best, and I will be forever grateful for her mentorship.

I would also like to thank the rest of my thesis committee: Andrew Appel, Sharad Malik, Michael Pellauer, and David Wentzlaff. Their feedback, suggestions, and mentorship made this thesis possible. I would especially like to thank Michael Pellauer for all of his support and guidance during my internships and during my time back at Princeton. Michael went out of his way to keep our collaborations going strong, and his input really helped my Ph.D. start to take shape. I'm extremely thankful that he was willing to make time for our weekly meetings and frequent emails, even through a change in employers.

I'd also like to thank all of the other members of the former VSSAD, especially Joel, Angshu, Aamer, Neal, Michael, Elliott, Bushra, and Mohit. Joel Emer was an excellent mentor to me during my three internships with the group, and I'm grateful for his support during and after my time there. I'm really looking forward to working with many of them again at NVIDIA.

Abhishek Bhattacharjee has been an amazing friend and mentor to me since the day I joined MRMGroup. From the beginning, he taught me what it means to be

a Ph.D. student and an academic, and I'm infinitely better off for having him as someone I can trust and count on for advice. I'm really glad we got to start working together again, and I hope there's more where that came from in the future.

In addition, I have to give huge thanks to all of the other members of MRMGroup: Sibren, Carole, Wenhao, Yavuz, Ozlem, Ali, Tae Jun, Logan, Elba, Caroline, Yatin, Themis, Shruti, Daeki, and Geet. They made it fun to be in the office every day, and we shared so many thoughts, laughs, lunches, insights, and stories that it's hard to even remember them all. I also have to give a shoutout to the Malik group members, the Wentzlaff group members, the reading group members, and everyone else with whom I've worked and become friends during my time at Princeton.

I've benefited tremendously from other collaborations and interactions as well. Thanks to Kevin Skadron, Kelly Shaw, and Ke Wang for their help during our earlier teleconferences. Thanks also to the staffs of the Electrical Engineering and Computer Science departments for helping me through the logistics of the program. I'd especially like to thank Lori Bailey and Stacey Weber Jackson for being so friendly and helpful with anything I ever asked of them.

A very special thank you goes to my wonderful girlfriend, Sunha Ahn. Since that first EE holiday party we attended, we shared so many special and important moments together, at Princeton and around the world. We made each other laugh, we supported each other throughout the bad times, and we celebrated together during the good times. She kept me motivated, and she kept me focused on what is really important in life. I couldn't have made it through my Ph.D. without her.

Finally, I would like to extend a huge thank you to everyone on both sides of my family. First and foremost, I have to thank my parents for channeling my interests in math and science into a successful career path, for making the sacrifice to send me to Loyola and to Penn, and for the endless and uninterrupted love and support. I also have to thank so many others: David, Mamá, Papá, Martha, Laura, John, Brad,

Reagan, Carmen, and the hundreds of other actual and honorary Cubans that were behind me all the way. On the other side, I would like to thank Grandpa, Marion, Josh, Patty, Ashley, Sharon, Jessica, Adam, and Bryan for being there for me as well. I am extremely lucky to have come from such a huge and loving extended family. I am also extremely lucky to have benefited from their decisions to escape bad situations across the world to make better lives for themselves and for their families here in America. Their love and their sacrifices allowed me to get where I am today, and for that reason, I dedicate this thesis to them.

To my parents.

Contents

Abstract iii

Acknowledgements vi

1 Introduction **1**

1.1 Motivation 1

1.2 Research Challenges and Goals 5

1.3 Defining Memory Models in an Architecture-Independent Manner . . 7

1.3.1 Dynamic Migration Between Consistency Models 8

1.4 Microarchitecture-Level Memory Consistency Models 9

1.4.1 Pipeline Models 11

1.4.2 Cache and Memory System Models 11

1.4.3 Address Translation 12

1.5 Thesis Contributions 12

1.6 Thesis Outline 13

2 Background and Related Work **15**

2.1 Basic Overview 15

2.1.1 From Sequential Consistency to Weak Ordering 16

2.1.2 A Stack of Memory Consistency Models 20

2.1.3 Coherence vs. Consistency 20

2.1.4 Litmus Tests 23

2.2 Defining Weak Memory Models 24
2.2.1 Motivating Example 24
2.2.2 Operational vs. Axiomatic Models 27
2.2.3 Defining Models Which Reflect Actual Hardware 29
2.2.4 Other Subtle Ordering Relationships 32
2.3 Empirical Analysis of Memory Models 37
2.3.1 Correctness 38
2.3.2 Performance 39
2.4 Comparing and Mapping Between Models 40
2.5 Simplifying Abstractions Made in This Thesis 43
2.6 Chapter Summary 44

3 Memory Ordering Specification Tables and ArMOR 45
3.1 Introduction 45
3.2 Motivating Example 47
3.3 Memory Ordering Specification Tables 51
3.3.1 Store Atomicity and Ordering Strength 51
3.3.2 Same-Address Dependencies 53
3.3.3 Fence Cumulativity 55
3.3.4 Summary 56
3.4 ArMOR: Comparing and Manipulating MOSTs 56
3.4.1 MOST Partition Refinement 56
3.4.2 ArMOR Comparison Operators 58
3.4.3 ArMOR Comparison Examples 60
3.5 Related Work 62
3.6 Chapter Summary 64

4 Dynamic Translation Between Memory Consistency Models 65

4.1 Introduction . . . 65

4.2 Motivating Example . . . 67

4.3 Cross-MCM Dynamic Binary Translation . . . 69

4.3.1 High-Level Operation . . . 69

4.3.2 Shim Finite State Machines: Overview . . . 70

4.3.3 Shim FSM Generation . . . 73

4.3.4 Shim FSM Generation Example . . . 78

4.3.5 Design Considerations . . . 79

4.4 Evaluation Methodology . . . 83

4.4.1 Pintool-Based Exploration Methodology . . . 83

4.4.2 Hardware Simulation Methodology . . . 85

4.5 Experimental Results . . . 87

4.5.1 Shims for a Broad Range of Scenarios . . . 87

4.5.2 Performance of Software Shim Implementations . . . 89

4.5.3 Performance of Hardware Shim Implementations . . . 92

4.6 Takeaways . . . 93

4.7 Related Work . . . 95

4.8 Chapter Summary . . . 96

5 PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models 97

5.1 Introduction . . . 97

5.2 Motivating Example . . . 99

5.2.1 Microarchitecture-Level Analysis . . . 101

5.2.2 Deriving Correctness from Microarchitecture-Level Axioms . . . 104

5.3 PipeCheck Microarchitecture-Level Analysis . . . 105

5.3.1 Microarchitecture Definition . . . 106

5.3.2 PipeCheck Model Specification Language 109
5.3.3 Transitivity of μhb Edges 115
5.3.4 Modeling Microarchitectural Optimizations 118
5.4 Verification Flow 120
5.4.1 Litmus Test-Based Verification 120
5.4.2 Automated Constraint Solver Algorithm 124
5.4.3 Software Implementation 129
5.4.4 Runtime 130
5.4.5 Caveats 130
5.4.6 Future Work: Formal Equivalence/Implication 131
5.5 Experimental Methodology 132
5.6 Results Across Litmus Tests 133
5.7 Case Studies 136
5.7.1 Speculative Load Reordering 137
5.7.2 Consistency Bug in gem5 O3 Pipeline 138
5.7.3 WeeFence 140
5.8 Related Work 141
5.9 Chapter Summary 144

6 CCICheck: Verifying the Coherence-Consistency Interface 145

6.1 Introduction 145
6.2 Motivating Example 148
6.2.1 Coherence-Consistency Interface Mismatches 148
6.2.2 The Window of Vulnerability Problem 150
6.3 The ViCL Abstraction 153
6.3.1 ViCLs: Definition and Usage 153
6.3.2 ViCL Timeline Example 155
6.3.3 Using ViCLs in μhb Graphs 157

6.4 CCICheck and μhb Graphs 159
6.4.1 ViCL-Aware Microarchitecture Definitions 160
6.4.2 Example: A Model with L1 Cache ViCLs Only 161
6.4.3 Multilevel Caches 167
6.5 Experimental Methodology 168
6.6 Case Studies 169
6.6.1 Partially-Incoherent Caches and L1 Cache Bypassing 170
6.6.2 Lazy Coherence 172
6.6.3 Window of Vulnerability/Peekaboo 175
6.6.4 Performance Results 178
6.7 Related Work 180
6.8 Chapter Summary 182

7 Conclusion and Future Directions **183**
7.1 Future Directions 183
7.1.1 Further Extensions of PipeCheck 183
7.1.2 Formal Correctness Proofs 185
7.2 Thesis Conclusions 186

A Gallery of MOST-based Definitions for Well-Known Architectures **190**

B Gallery of Shim FSMs **195**

Bibliography **249**

Chapter 1

Introduction

1.1 Motivation

Over the second half of the twentieth century, general-purpose computing developed from a theoretical exercise into a practical and ubiquitous reality. In 1936, Alan Turing invented what is now called a universal Turing machine, an idea which is now widely considered as the theoretical basis for the modern notion of a computer [Tur37]. ENIAC, cited by many as the world's first fully-digital general-purpose computer[1], was publicly unveiled in 1946 [EM]. The first stored-program computer, the Manchester Small-Scale Experimental Machine, ran its first program in 1948 [Cen15]. The first commercially-available general-purpose digital computer, the Ferranti Mark 1, was purchased in 1951. By the end of the century, fundamental technology advances such as the bipolar transistor, the integrated circuit, CMOS logic, and corresponding advances in software and design technologies drove computing from being a niche application into the commodity that it remains today.

The first half-century of digital computing followed a number of remarkable trends. Gordon Moore, based on a few early observations, predicted in 1965 that transistor

[1] Other competitors for this or similar claims include the Atanasoff-Berry Computer, the Colossus Machine, and the Z3 [Com15]. I acknowledge my personal bias towards ENIAC, if only because I received by bachelor's degree from the University of Pennsylvania, where ENIAC was created.

density would double roughly every two years [Moo65]. This phenomenon, popularly dubbed Moore's Law, combined with improvements to the architectures of the processors themselves to lead to a doubling of performance roughly every 18 months. Likewise, Dennard et al. observed in 1974 that by scaling supply voltage downwards with transistor area, transistor power density would be roughly constant [DGY+74]. Dennard scaling, together with Moore's law, allowed for high levels of integration, giving high performance at manageable power. Although these trends were generally extrapolated from no more than a few data points, each remarkably held true for decades, leading them to be popularly classified as "laws" and to guide the progress of the semiconductor industry for nearly half a century.

Unfortunately, in the early 2000s, computing hit the power wall [BC11, KM08, SKM15]. Dennard scaling started to break down around 2005, as the power consumption and thermal dissipation requirements of increasingly smaller and faster transistors became impossible to satisfy. This breakdown brought about an end to the exponential growth in single-core performance seen during the previous decades, and it drove the industry to shift to *multicore processors* as a way to deliver continued processor performance scaling [ABC+06, HM08]. The ability to extract and make efficient use of parallelism hence became a prerequisite for making the most use of the hardware. Multicore processors are now found throughout most major segments of the industry, from mobile devices to laptops and desktops to supercomputers.

As a consequence of the paradigm shift to multicore processors, *shared memory* parallelism has taken on renewed importance. While software parallelism models can (and do) vary widely, at a hardware level, the predominant paradigm is for the cores in a multicore processor to synchronize and communicate data by reading and writing into a shared memory space. This communication and synchronization takes place according to the *memory consistency model*: the set of rules specifying how and when accesses made by each core become visible to other cores. Unfortunately, the process

of building consistency models and synchronization primitives that are correct, high-performance, and easy to use has proven notoriously difficult over the decades, and so memory models remain the subject of ongoing research even today.

Lamport defined sequential consistency (SC), the first explicit memory consistency model, in 1979 [Lam79]. Sequential consistency (then and today) represents the most intuitive model of shared memory concurrency. It states, roughly, that concurrent programs behave as if they were a single-threaded program formed by interleaving the instructions from each thread. SC was notable for formalizing the rules that many programmers were implicitly assuming about the behavior of parallel programs, particularly in light of the fact that even then, concurrent hardware did not always enforce SC. Today, most computer architects consider the performance of sequentially consistent processors to be too low to be commercially competitive. In modern processors, the requirements of sequential consistency are routinely violated by hardware optimizations such as memory access reordering and store buffering [AMD13, ARM13, IBM13, Int10, Int13a].

In 1986, Dubois et al. proposed the idea of defining a *weak memory model*—one which is weaker (i.e., more permissive) than SC [DSB86]. Weak memory models present a different (usually more complicated but higher-performance than SC) set of rules for shared memory communication. In this approach, the burden of building correct communication and synchronization primitives is shifted onto the programmer and/or the programming language, as either or both must be able to understand and reason about the increasingly complicated set of rules governing cross-core communication.

In spite of the significant added complexity that weak memory models bring, nearly all modern multicore processors implement some form of weak memory model. The shift of computing into the multicore era drove a renewed practical interest in memory consistency model research in the 2000s, and this led to efforts to im-

prove and formalize the memory models used by C/C++ [BA08], Java [MPA05], x86(-64) [OSS09a], and Power [AMT14, MHMS+12, SSA+11], among others. However, numerous challenges remain. On one hand, memory model analyses following a formal mathematical approach are sufficiently challenging that they tend to take place on the timeframe of years, rendering them unable to fully adjust to the fast-moving world of computing. On the other hand, with every advance in the scientific understanding of memory models comes another discovery that some existing model or framework is broken, thereby revealing just how challenging these problems are [ABD+15, ND13, VBC+15].

Although memory models have been studied for decades, two contemporary trends in computer architecture are further increasing the difficulty of specifying and reasoning about memory consistency models. First of all, many modern systems are *architecturally heterogeneous*—they contain processor cores and/or other components which have varying instruction sets and therefore, usually, varying memory consistency models. The GPGPU paradigm, in which a graphics processing unit (GPU) performs general-purpose (GP) computation in collaboration with a CPU, is one notable example, but mobile systems-on-chip (SoCs) may contain as many as a half dozen different components with different instruction sets and MCMs [Qua15, Tex10]. Second, the traditional abstractions of programming language, architecture, and microarchitecture are becoming increasingly blurred. Intermediate representations such as Java bytecode and LLVM IR are increasingly commonly being used as semi-permanent representations of code, and "virtual instruction sets" such as NVIDIA PTX [NVI13b] and HSAIL [HSA13] are diminishing the dominance of any one particular hardware instruction set. Concurrent with this trend is an increase in the popularity of dynamic just-in-time (JIT) compilers [NVI13a, Khr], optimizers [NVIb], and binary translators [DVT12, VT14], cases for which the memory consistency implications have not been as widely studied. These two trends show the importance of having memory

consistency model analysis techniques which are flexible enough to adapt to such rapidly-changing conditions.

This thesis bridges the gap between the benefits of rigorous formalism and the need for practical applicability in memory model analysis, all while addressing the recent trends discussed above. The next section describes some of the key challenges that arise in this space, and it lays out the specific goals for this thesis.

1.2 Research Challenges and Goals

The past three decades have shown that specifying and reasoning about weak memory models is extremely difficult and prone to numerous pitfalls and unexpected corner cases. In fact, it remains a challenge even to precisely and completely define models such as C++, Java, ARM, or Power which are nevertheless in widespread use. Furthermore, since the software memory model needs to be implemented in terms of the hardware memory model of the processor on which it will execute, the job of building tools such as compilers in a way that correctly accounts for memory models is itself a significant additional challenge [BMO+12, VBC+15].

Existing approaches and models are insufficient in various ways. For example:

- They are imprecise: many industry specifications are written in natural language and are often too vague to be used to analyze certain cases [AMD13, ARM13, IBM13, Int13a, NVI13b].

- They are incomplete and/or incompatible: e.g., there are cases in which various recent formalizations of the Power memory model disagree with each other [SSA+11, MHMS+12, AMT14].

- They are impractical and/or inaccessible: e.g., the Java memory model is considered by many to be difficult to use in practice due to its "replay" approach to deriving valid executions [PVJ15].

- They are unsound: e.g., numerous early iterations of the C++, Java, and Power memory models were shown to disagree with the realities of the underlying implementations [Alg12, BOS+11, SSA+11, ŠA08] and each model continues to be refined even today [Bat04, BMO+12].

As a result of the above shortcomings, it remains difficult to analyze the memory consistency model implications of building new microarchitectures, toolchains, and programming languages. Even when model definitions exist, they may not be practical to use. For one thing, nearly all existing models are, by necessity, abstracted somewhat from the full realities of the microarchitecture. For another, user-friendly tools which allow non-experts to analyze the memory model behavior are often either non-existent [MHMS+12] or prone to state space explosion [SSA+11]. As a result, determining correct and optimal mappings for critical standard data structures onto weak memory models remains an active area of research even today [BMO+12, LGCP13, LPCZN13].

An additional ongoing challenge is to ensure that hardware being built and shipped to customers is correct with respect to its specification. There have been numerous (in)famous examples of bugs appearing in all parts of the processor, and the memory system is no exception. Although the memory consistency model is more precisely and formally defined than many other components of the processor design, the tools which allow these formalisms to be applied to practical microarchitectural use cases are still not sufficiently developed. This fact leads bugs such as those in implementations of transactional memory [Int15], load-load reordering [ARM11, Elv], or fences [ABD+15] to continue to appear in hardware even today. The trend towards increased architectural heterogeneity and the blurring of abstractions in the hardware-software stack will only serve to make these problems more challenging.

In response to the above challenges, this thesis focuses on specifying, verifying, and translating between memory consistency models. It does so in a way that bridges

the gap between formal analysis methods and practical microarchitectural use cases. In particular, it applies rigorous methodologies to the analysis of microarchitecturally-important cases such as out-of-order execution, speculative reordering, partially incoherent caches, virtual-to-physical address translation, and coherence protocols which violate the formal specifications of coherence. It also builds practical tools which can be used to analyze both currently relevant and forward-looking use cases. The rest of this chapter outlines the approach taken to overcome the challenges and to solve the goals described above.

1.3 Defining Memory Models in an Architecture-Independent Manner

One shortcoming of existing approaches to defining memory models is that in general, two unrelated memory models cannot easily be directly compared. In the best case, the two models may each be specified using some precise formalism, but unless the two formalisms use matching approaches and notations, it is difficult or impossible to simply take a component from one model and map it into the language of the other. Instead, the process of deriving a correct mapping between two models generally requires either a conservative over-approximation or a custom third formalism consisting of manual proofs that often take years to complete [BMO$^+$12, LPCZN13, SMO$^+$12]. This fact makes it difficult to reliably build compilers, dynamic binary translators, or any other mechanism which has to perform such cross-model mappings.

Chapter 3 presents the Memory Ordering Specification Table (MOST), an architecture-independent and model-independent specification format for memory models [LTPM15a, Lus15]. MOSTs provide a precise, self-contained notation for defining the memory ordering requirements or enforcements of the various individual components of a memory model. MOSTs describe preserved program order (PPO,

i.e., the set of orderings enforced by hardware by default), fences, inter-instruction dependencies, and all other forms of ordering mechanism used by a model. Taken together, the MOSTs for each individual component collectively form the definition of the memory model in question.

The benefit of the MOST specification format is that by providing a common model-independent language, any two MOSTs can be compared to each other, even when the MOSTs originate from unrelated memory models. In addition to MOSTs themselves, Chapter 3 also presents the ArMOR framework, a primitive "arithmetic" on MOSTs which can be used to analyze, compare, and manipulate MOSTs. ArMOR enables MOSTs to be flexibly applied in compilers, dynamic translators, and other interesting use cases. The ArMOR framework is released as an open-source tool [Lus15].

1.3.1 Dynamic Migration Between Consistency Models

A key benefit of the ArMOR framework is that it enables new memory consistency-related use cases which were not previously feasible to build. Chapter 4 highlights the example of dynamic binary translators (DBTs) that translate between different instruction set architectures (ISAs) [LTPM15a]. Although cross-ISA binary translation has been shown to deliver performance and/or energy benefits, existing frameworks simply do not address the memory consistency implications of doing so [DVT12, VT14]. Likewise, existing architectural emulators such as the Android emulator and QEMU implement single-threaded backends in order to simply avoid the memory model issue altogether [Goo15, Qem15].

Chapter 4 presents an in-depth case study of using MOSTs and the ArMOR framework to build memory-model-aware dynamic binary translators. Given the set of MOSTs defining the source and target memory models, this chapter presents an algorithm for automatically deriving *shims*, or small, self-contained translation mod-

ules which automatically insert fences or other ordering mechanisms as needed. This chapter demonstrates that both software and hardware shim implementations have low to negligible performance overheads, making them practical to implement. The appendix to this thesis also presents a gallery of shims for different scenarios. The code used to generate this appendix is open source as well [Lus15]. The chapter concludes by considering extended use cases such as optimization via dynamic removal of redundant fences and by presenting a broader view of the ways in which future architectures can make the analysis and implementation of such technologies easier.

1.4 Microarchitecture-Level Memory Consistency Models

Even though hardware memory models are intentionally abstracted from the details of any one particular implementation, they are often nevertheless inspired by some class of microarchitectural features. For example, microarchitectural store buffering inspired the design of processor consistency and SPARC total store ordering (TSO), and some models contain explicit references to microarchitectural features such as branch prediction [SSA+11, PVJ15]. Nevertheless, the models are intentionally designed to be portable across varying implementation for the sake of backward/forward compatibility: a memory model may be implemented by more than one company (e.g., TSO by SPARC/Oracle [SPA94b], Intel [Int13a], AMD [AMD13], and others) and/or by more than one chip produced by the same company (e.g., Intel Haswell/Broadwell "big cores", Bonnell/Silvermont "small cores"). This fact means that by design, existing models cannot specify behaviors specific to any one implementation. As a result, existing frameworks cannot generally be applied either to microarchitectural verification or to scenarios such as modern GPUs which do not have well-defined architecture-level memory model specifications.

Chapter 5 introduces PipeCheck, a methodology and open-source tool for verifying the correctness of a given microarchitecture against a given architecture-level memory model specification [LPM14, LPM15, Lus14]. PipeCheck provides a domain-specific language (DSL) for defining a multi-event axiomatic model of the memory consistency behavior of one particular microarchitecture. It uses this model to quickly and automatically identify cases in which the microarchitecture may be stricter than necessary (e.g., an SC implementation of TSO) or weaker than necessary (e.g., due to an implementation bug).

Specifically, PipeCheck defines the memory ordering behavior of a pipeline using a set of microarchitecture-specific ordering axioms: some defining per-instruction behavior, some defining per-pipeline stage behavior, and some defining other relevant ordering enforcement behaviors. For example, an axiom may state (using more formal language) that "the decode stage is FIFO" or that "a load may either take a cache hit, take a cache miss, or forward from the store buffer". PipeCheck then determines whether any executions which violate the architecture-level specification are observable in terms of the axioms provided. The axioms are designed with the intention that, to the extent possible, they can be verified and analyzed entirely independently. In this way, the PipeCheck methodology as a whole therefore reduces the burden of verifying a particular pipeline as a whole to the more tractable problem of verifying the localized behaviors of individual pipeline stages or events.

There are numerous microarchitectural components which play a role in memory ordering: the pipeline, the cache hierarchy, the address translation mechanisms, the networks-on-chip (NoCs), and so on. The analysis of various microarchitectual subsystems using the PipeCheck methodology is organized as described below.

1.4.1 Pipeline Models

Chapter 5 focuses on the role of the pipeline itself in enforcing the memory consistency model [LPM14, LPM15]. The memory hierarchy itself often provides little to no ordering guarantees beyond some form of *cache coherence*, leaving the pipeline largely responsible for enforcing consistency guarantees. Likewise, fences or other explicit ordering mechanisms are generally enforced within the pipeline itself; such mechanisms do not often propagate explicitly to the memory system. Nevertheless, the pipeline may itself contain a number of performance optimizations which may themselves introduce weak memory behavior. Chapter 5 therefore demonstrates how to verify pipelines containing optimizations such as of out-of-order execution, speculative load reordering, or other similar mechanisms.

1.4.2 Cache and Memory System Models

While Chapter 5 abstracts away much of the behavior of the cache hierarchy, Chapter 6 highlight ways in which specific low-level behaviors of certain cache hierarchies may need to be addressed explicitly [MLPM15]. For example, although most architecture-level memory models contain some formal notion of coherence, there are cases in which particular cache coherence protocols may in fact violate certain interpretations of the word "coherence", as "coherence" is defined in varying ways by different authors. Adding an explicit model of the coherence-consistency interface (CCI) to PipeCheck allows it to verify that the microarchitecture as a whole continues to implement the memory model correctly, given the precise behavior of the coherence protocol.

CCICheck extends the methodology and software implementation of PipeCheck to explicitly model the behavior of a particular cache coherence protocol. CCICheck provides the key innovation of the *value-in-cache-lifetime* (ViCL), an abstraction of the lifetime of a given piece of data being held in a particular cache line in some spe-

cific cache. The ViCL abstraction allows CCICheck to explicitly and scalably model behaviors such as cache hits vs. cache misses, incoherent caches, and lazy cache invalidation, and these models allow CCICheck to verify the correctness of each behavior with respect to the memory model. The chapter concludes with an exploration of how CCICheck can be used to verify the correctness of various coherence protocols which have been recently proposed in the academic literature.

1.4.3 Address Translation

A related but less-studied question is the role that virtual-to-physical address translation plays in the specification and verification of memory consistency models. Romanescu et al. [RLS10] brought attention to the fact that problems such as TLB synonyms—two virtual addresses which map to the same physical address—can imply ordering requirements that cannot be inferred from virtual or physical addresses alone. Section 7.1.1 discusses ongoing work to apply PipeCheck to the verification of such scenarios [LSMB15].

1.5 Thesis Contributions

This thesis makes an impact via the following significant contributions:

- Inspired by the shortcomings of existing informal memory consistency model specification techniques, this thesis defines the Memory Ordering Specification Table (MOST), a precise, self-contained notation for defining components of a memory consistency model [LTPM15a]. MOSTs improve upon existing specification formats by directly including features (e.g., fence cumulativity and degree of store atomicity) that are left unaddressed by simpler models. The key contribution of MOSTs is that they eliminate much of the imprecision found in

typical industry definitions of memory consistency model behaviors and hence also of the analyses that make use of such definitions.

- To demonstrate the practical value of MOSTs, this thesis also presents the open-source ArMOR framework for systematically analyzing and manipulating MOSTs [Lus15]. The precision and completeness of MOSTs allows them to be analyzed to improve the rigor of use cases which already exist (e.g., compilers) and in use cases which are forward-looking (e.g., cross-ISA dynamic binary translation). This thesis performs an in-depth analysis of the latter, showing not only that the necessary analysis can be automated using ArMOR, but also that such translation can be made practical in a variety of hardware and software scenarios.

- The work in this thesis is the first to present a general-purpose formal axiomatic memory model framework for specifying and verifying microarchitectural memory access reordering behavior [LPM14, LPM15]. While previous work had shown the need for rigor and formalism when analyzing memory consistency models at the architecture or software level, no previous work had applied such techniques to the microarchitecture space. PipeCheck's microarchitectural happens-before (μhb) graphs and the domain-specific language (DSL) that defines how they are enumerated provide a clear, visualizable approach for specifying and verifying implementation-level memory reordering optimizations.

1.6 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 presents a background of history and current practice of memory consistency models. Chapter 3 introduces the MOST specification format and the ArMOR framework for systematically manipulating MOSTs. Chapter 4 then presents an in-depth case study of the use of MOSTs

to perform flexible and practical cross-ISA dynamic binary translation. Chapter 5 introduces PipeCheck, a methodology and automated tool for defining and analyzing microarchitecture-level consistency models, while focusing on the behavior of the pipeline itself. Chapter 6 then develops a microarchitecture-level model for verifying the consistency implications of cache coherence protocols. Finally, Chapter 7 presents ongoing and future research directions and then concludes this thesis.

Chapter 2

Background and Related Work

This chapter presents an overview of past and ongoing research in the field of memory consistency models. This overview serves as the foundation for the rest of this thesis. Section 2.1 presents high-level background on memory consistency models in general, and Section 2.2 describes the approaches most often used to define weak memory models. Section 2.3 explains how empirical methods are used to complement formal analyses in practical usage scenarios. Section 2.4 discusses some challenges faced when trying to compare or map between different memory models. Section 2.5 describes some simplifying assumptions that we make in this thesis, in keeping with other related work on memory consistency models. Section 2.6 then concludes.

2.1 Basic Overview

A memory consistency model is defined by Adve and Gharachorloo as follows [AG96]:

> [A] memory consistency model [...] provides a formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by a system. Effectively, the consistency model places

restrictions on the values that can be returned by a read in a shared-memory program execution.

The most intuitive way to define a memory model is to state that each load is required to return the value of the latest store to the same address. This works in single-threaded code, because "latest" can be defined as the most recent such store in *program order*: the order in which the instructions were originally laid out in the code being executed. In multithreaded programs, however, accesses from other cores may be interleaved with the accesses of the core in question. This means that there will in general be many values that any given load might plausibly return. The goal of a memory model is therefore to specify which values are legal to return and which are not.

A naive approach to defining the multithreaded sense of "latest load" is to choose the store (originating from any core in the system) that has happened most recently in an absolute physical sense. This is sometimes known as *strict consistency*. However, global synchronization or coordination among a distributed set of cores is difficult or even impossible to implement in practice, as any inter-core communication or synchronization requires passing of messages which themselves take non-trivial (and often unpredictable) amounts of time to propagate. Strict consistency is therefore considered infeasible to implement in practice. In practical models, "latest" must therefore be defined in such a way that it does not refer to any notion of a global physical timeline.

2.1.1 From Sequential Consistency to Weak Ordering

The first memory consistency model, *sequential consistency* (SC), was defined in 1979 by Lamport [Lam79]. SC was originally defined in terms of two properties:

1. The result of any execution is the same as if the operations of all the threads were executed in some sequential order.

2. The operations of each individual thread appear in this sequence in the order specified by the thread.

The key benefit of SC is that it closely matches programmers' intuitions about how multithreaded programs should behave. Under SC, for each load, the set of values that can be returned consists of exactly one element: the most recent store to the same address, according to the interleaving implied by rule 1.

In spite of its intuitiveness, Lamport noted in his paper that enforcing sequential consistency "may not be worth the price of slowing down the processors" [Lam79]. In fact, nearly all modern systems intentionally sacrifice the elegance of SC to deliver higher performance. Memory accesses may be *reordered* from how they originally appear in the code into an order that improves latency and/or throughput, thereby violating rule 2 of SC. Furthermore, accesses may be *buffered* in ways that make them visible to certain threads before they become visible to other threads. This causes different threads to potentially observe the same set of events occurring in different orders, thereby violating rule 1 of SC.

Weak memory models legalize the reordering of memory accesses, buffering of memory acceses, and lack of global consensus. Weak models have been defined in various ways over the years. Dubois et al. first defined them as follows [DSB86]:

> In a multiprocessor system, storage accesses are weakly ordered if 1) accesses to global synchronizing variables are strongly ordered and if 2) no access to a synchronizing variable is issued in a processor before all previous global data accesses have been performed and if 3) no access to global data is issued by a processor before a previous access to a synchronizing variable has been performed.

Here, synchronizing variables are variables which are explicitly declared as such by the programmer. Strong ordering is a concept that has been found to be too weak;

most modern definitions replace "strongly ordered" in the above definition with "sequentially consistent". Finally, "perform" is defined by the authors as follows:

> A LOAD by processor I is considered performed with respect to processor K at a point in time when the issuing of a STORE to the same address by processor K cannot affect the value returned to processor I. A STORE by processor I is considered performed with respect to processor K, at a point in time when an issued LOAD to the same address by processor K returns the value defined by the STORE. An access by processor I is performed [(or globally performed)] when it is performed with respect to all processors.

Some memory models use the same definition of "perform" to this day [IBM13]. However, it has proven difficult to formalize, as the loads and stores to which the definition refers are hypothetical: they may not (and in general do not) actually exist, thereby making it hard for formal analyses even to refer to them. Chapter 5 addresses this problem and presents a microarchitecturally-inspired solution.

In response to the above shortcomings in the Dubois et al. definition, Adve and Hill proposed an alternative definition of weak ordering that by design focused on outcomes rather than mechanisms [AH90]:

> Hardware is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all software that obey[s] the synchronization model.

To go along with this definition, they also propose the now widely-used *data-race-free* (DRF) family of synchronization models. DRF rules state that all conflicting accessses (i.e., accesses from different cores to the same address and for which at least one is not a read) must be ordered by a happens-before relationship induced by explicit synchronization operations on the cores in question.

In modern informal usage, any memory model which violates either of the two rules of SC may be considered "weak" or "relaxed". The Adve and Hill definition of weak ordering focuses on the ability of a processor to restore sequential consistency, and most modern architectures do satisfy this requirement. However, modern usage frequently diverges from the Adve and Hill definition in two key ways. First, their definition technically requires synchronization operations to access memory, while modern hardware more commonly performs synchronization using fences or other mechanisms which do not themsleves access memory. Second, and more fundamentally, a large body of modern research in memory models explicitly focuses on behaviors which are *not* sequentially consistent as the explicit area of interest, even though these behaviors are explicitly forbidden under the Adve and Hill definition [Adv93]. Essentially all modern hardware uses a model weaker than SC, and many performance-critical data structures are carefully and explicitly optimized to take advantage of the performance benefits of non-SC behavior [BMO+12, LGCP13, LPCZN13, MRP+14]. This thesis primarily falls within this context of properly specifying and verifying behaviors which are desirable even though they are not sequentially consistent.

Even from the beginning, Lamport cautioned that when designing synchronization protocols "at the lowest level of the machine code" (i.e., on non-SC machines), "verifying their correctness becomes a monumental task" [Lam79]. Today, weak memory models are often considered highly counterintuitive and difficult to work with, and software models often make it a goal to hide such behavior from the user when it does exist [BA08, MPA05]. Nevertheless, since non-SC behavior is in widespread use at both the hardware and the software levels, a proper understanding of weaker-than-SC memory models is crucial to the correctness of modern systems.

Layer	Notable Modern Examples
Software	C++ [BA08], Java [MPA05]
Intermediate Representation	HSAIL [HSA13], LLVM IR [LA04], NVIDIA PTX Virtual ISA [NVI13b]
Architecture	SC [Lam79], x86-TSO [OSS09a], IBM Power [AMT14, MHMS+12, IBM13, SSA+11]
Microarchitecture	(this thesis)

Figure 2.1: A stack of memory models. Each builds on top of the layers below it.

2.1.2 A Stack of Memory Consistency Models

Just as in many other computing disciplines, memory models are generally built in multiple layers. Hardware-level memory models provide a layer of abstraction across different microarchitectural implementations. Software-level memory models then provide a layer of portability across different hardware models. Other layers (e.g., compiler intermediate representations) may be used as well. Figure 2.1 provides some examples relevant to systems in use today. Each layer must be implemented in terms of the layer below it in the stack. Note that much of this complexity is often hidden inside compilers, libraries, or other similar tools.

Prior to this thesis, most research into formally specifying and analyzing memory models took place at the architecture level and above. This left a large gap between the high-level architecture specification and the implementation level, and verification within this gap could not be easily performed. As such, this thesis focuses on specifying and analyzing architecture- and microarchitecture-level memory models in an effort to bridge this gap.

2.1.3 Coherence vs. Consistency

Cache coherence is a property that is distinct from yet closely related to and often confused with memory consistency. In this section, we summarize the competing def-

initions of coherence used by various authors, identify how they relate to consistency, and then state the terminology we will use in the rest of this thesis.

Coherence is defined in many varying ways by different authors; no single definition can be considered universal. One common definition is the property that the accesses to each individual address are sequentially consistent [CLS03]. Many authors use the narrower property that there exists a globally-agreed-upon total ordering on the visibility of stores (but not loads) performed to each address [Alg12, SSA+11]. This particular ordering is called *write serialization* (`ws`). Cache coherence protocols often define coherence in terms of a pair of properties: the Single Writer/ Multiple Reader (SWMR) property says that there may be zero or more read-only copies or exactly one read-write copy (but not both), and the Get Latest Value (GLV) property states that reads to each address return the value of the latest write to that address [SHW11]. Other authors use still other subtly distinct definitions [HP11, MHMS+12, MHW03, SSA+11]. To avoid confusion arising from competing definitions of coherence, the term "coherence" is not used in this thesis without any other context. Instead, preference is given to more precisely-defined properties such as write serialization, SWMR, and so on.

Perhaps surprisingly, the role of coherence in memory consistency model specifications is still partially unsettled. Nearly all consistency models for multicore CPUs and heterogeneous SoCs (e.g., those with CPUs and GPUs) explicitly require coherence [AMT14, HSA13, MHMS+12, NVI13b, OSS09a, SSA+11]. In software models, however, the role of coherence is less clear. C++11 did not originally address coherence explicitly [BA08]. However, this omission was later found (during a subsequent academic formalization process) to make the specification unsound [Bat04, BOS+11]. C++ now requires coherence in the form of four axioms relating the "happens-before" ordering to the "modification order" [ISO11a]. The current Java memory model does not mention coherence. At the same time, it has not yet been formally analyzed to

the same degree as the C++11 model has, and so it is possible that future revisions may need to include it as well.

Many architects consider coherence and consistency to be entirely decoupled concepts [Mar05]. According to this point of view, the job of the cache coherence protocol is simply to enforce orderings between accesses to the same address (i.e., coherence, for some definition of coherence) by correctly responding to requests and/or by stalling requests when necessary. The job of the consistency model is then to enforce orderings between accesses to different addresses. The key benefit of decoupling coherence and consistency is that verification of each part independently becomes dramatically easier. This is reflected in the literature: there are numerous papers on verifying coherence protocols [McM01, SSH+13, SKA13, ZLS10, ZBES14] or on verifying consistency while assuming a working coherence protocol [Alg12, AMT14, OSS09a, SSA+11], but there are far fewer on verifying coherence and consistency together [GSSVD00, TDF+02]. However, many consistency model implementations are tightly interwoven with the coherence protocol implementation to enable performance optimizations such as speculative load reordering [GGH91]. At this level, coherence and consistency inherently cannot always be decoupled, and as Chapter 6 shows, the details of individual coherence protocols can play an important role in enforcing coherence.

The question of how coherence relates to consistency comes up repeatedly in this thesis. Chapter 3 introduces a notation for precisely specifying the orderings that are or are not included in "coherence" for a variety of architectures. Chapter 5 uses an idealized model of memory which enforces only write serialization, but Chapter 6 dives deeper and carefully explores how consistency model implementations depend on write serialization, SWMR, and many other varying properties which may or may not be enforced by any given coherence protocol.

Thread 0	Thread 1
st [x]←1	ld [y]→r1
st [y]←1	ld [x]→r2
Proposed Outcome: 1:r1=1, 1:r2=0	

(a) Architecture-independent version

Thread 0	Thread 1
MOV [EAX], $1	MOV ECX, [EBX]
MOV [EBX], $1	MOV EDX, [EAX]
Proposed Outcome: 1:ECX=1, 1:EDX=0	

(b) x86-specific version

Figure 2.2: Litmus test `mp`.

2.1.4 Litmus Tests

Litmus tests are very small programs used to test and/or reason about the behavior of a memory consistency model. Figure 2.2 gives one example. A typical litmus test contains a small number of threads (generally 2-4) each executing a small number of instructions (generally 1-6). The instructions in a litmus test generally consist of explicit loads, stores, fences, or atomic read-modify-write operations. Non-memory instructions are generally omitted, except where relevant to, e.g., establish the presence of an address, control, or data dependency. The initial state of memory is, by convention, that all addresses start holding the value zero unless otherwise explicitly stated. Litmus tests also specify a particular outcome of interest. This outcome specifies the values returned by loads and/or the final value stored at a particular address. If the test is associated with a particular memory model, the test may also specify whether the proposed outcome is *permitted* or *forbidden* by the rules of that model. Such an annotation would serve as a reference during testing.

The instructions in a litmus test are often presented in a way that makes them independent of any particular architecture. In particular, litmus tests often abstract away details of instruction sets, use notation loosely, and so on. This allows the litmus test to focus on the memory model itself, as opposed to the irrelevant details of any one instruction set. It also in many cases allows for one test to be analyzed across a variety of memory models.

Litmus tests are used in various ways. First of all, litmus tests are used as a means of allowing humans and computers to reason about a particular behavior of interest. Second, they can provide a concrete complement to definitions which are otherwise incomplete and/or highly abstract. I will use litmus tests in both ways throughout this chapter and the rest of this thesis.

2.2 Defining Weak Memory Models

Driven by the need for higher-performance alternatives to SC, much modern research and development into memory consistency focuses on the specification and implementation of weak memory models. Unfortunately, as Lamport so presciently predicted, reasoning about weak memory models (i.e., determining which values may be legally returned by memory loads) is made significantly more difficult by the reordering, buffering, and lack of global consensus described earlier. Here, I provide some background on why these weakly-ordered implementations do not lend themselves to simple descriptions or specifications. Chapter 3 then introduces ways in which memory models can be described while taking these important subtleties into account, and Chapters 5 and 6 then describe how to rigorously analyze this buffering and reordering at the microarchitecture level.

2.2.1 Motivating Example

The most prominent property of weak memory models is that, in contrast to rule 2 of sequential consistency, they allow certain program order relationships to be relaxed. The specification of which program order relationships must be globally maintained is called *preserved program order* (PPO). In other words, if (a) comes before (b) in program order, and a→b must be preserved, then (a) must happen before (b) from

	Loads	Stores
Loads	✓	✓
Stores	— (`mfence`)	✓

Legend:	
✓	Ordering preserved
—	Ordering not preserved
(`mfence`)	Ordering preserved by `mfence`

Figure 2.3: One common (yet incomplete) definition of the total store ordering (TSO) memory model

Thread 0	Thread 1
`(i1) St [x]←1`	`(i4) St [y]←1`
`(i2) Ld [x]→r1`	`(i5) Ld [y]→r3`
`(i3) Ld [y]→r2`	`(i6) Ld [x]→r4`
Proposed Outcome:	
`0:r1=1, 0:r2=0, 1:r3=1, 1:r4=0`	

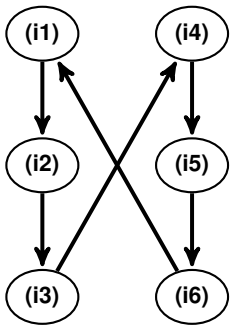


Figure 2.4: Litmus test `iwp2.4` demonstrates that proper memory models to be able to both capture and reason about very subtle behaviors

the point of view of every observer in the system. Otherwise, if a→b need not be preserved, some cores may see the effect of (b) before they see the effect of (a).

Preserved program order is often loosely specified using tables such as the one in Figure 2.3. Every cell such a table indicates whether the corresponding types of accesses in program order must be preserved. In this example, Total Store Ordering (TSO) allows for store→load reordering, but it forbids other types of reorderings [OSS09a, SPA94b]. The inspiration for TSO is that it allows for the insertion of store buffers between the core and the memory. These store buffers allow loads to bypass earlier stores, thereby improving overall latency. Preserved program order exists across a wide spectrum. Under sequential consistency, all program order relationships must be preserved. Under models such as those used by ARM and Power, few orderings must be preserved, but describing the orderings can be very difficult [AMT14].

Consider litmus test `iwp2.4` in Figure 2.4. This particular test is designed to highlight how store buffering introduces behavior that is non-trivial to characterize

and therefore requires non-trivial memory model specification approaches. The focus on store buffering can be seen from the choices of instructions in the threads of the test: because (i1) and (i2) are a write followed by a read to the same address, it is likely (but not strictly required) that (i2) will forward its data from (i1) while (i1) is still in the store buffer. Similarly, (i5) will likely forward from (i4).

Naive happens-before analysis would proceed as follows. Since (i1) is before (i2) in program order and (i2) reads from the value written by (i1), (i1) must happen before (i2). Likewise, (i4) must happen before (i5). If (i4) were to happen before (i3), then (i3) would be forced to return the value 1, contradicting the proposed outcome. Therefore, (i3) must happen before (i4). For similar reasons, (i6) must happen before (i1).

The `iwp2.4` litmus test outcome is not legal under sequential consistency, as there is no single ordering in which 1) program order is respected, and 2) all of the above ordering relationships hold. This is depicted by the cycle shown in Figure 2.4. However, in spite of the cycle, and in spite of what Figure 2.3 might imply, the outcome proposed in Figure 2.4 is permitted under Total Store Ordering (TSO). If (i2) and (i5) store buffer forward from (i1) and (i4), respectively, then the edges from (i1) to (i2) and from (i5) to (i6) are effectively "weaker" than the others (in a formal sense), and the cycle loses its effect. Therefore, this type of weakening must be taken into account by the analysis mechanism for TSO (and likewise for other models).

Since naive specification approaches are insufficient, a number of different approaches have been developed in the academic literature and put into use in practice. The next section describes some of these more advanced techniques, and we return to litmus test `iwp2.4` in Section 5.3.3.

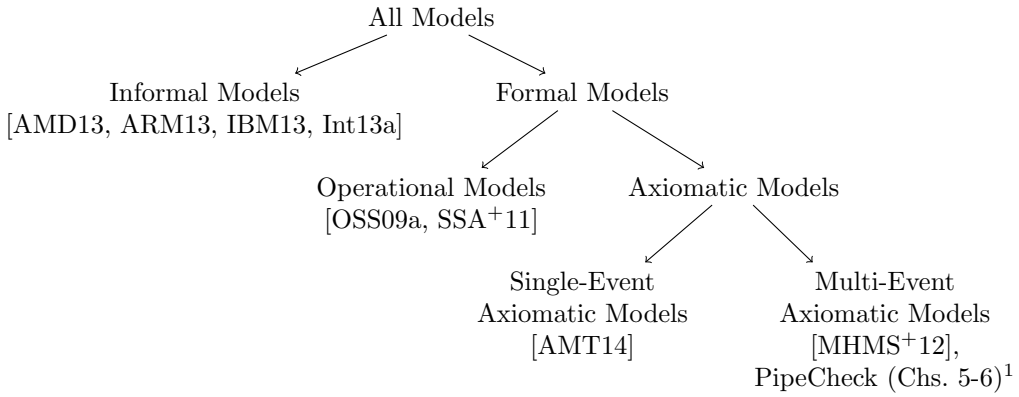


Figure 2.5: A taxonomy of common approaches used to define hardware memory consistency models

2.2.2 Operational vs. Axiomatic Models

Given that overly-simplistic definition approaches are insufficient to capture the behavior of real-world processors, a variety of more sophisticated approaches are used in practice. Figure 2.5 presents a taxonomy of approaches that are commonly used to define hardware memory consistency models. These approaches apply varying degrees of formality. Many models, from industry in particular, are described informally using natural language descriptions, specific examples, and/or explicitly enumerated rules. More recently, there has been a surge of interest in defining memory models using more formal mathematical notation which is amenable to being analyzed using rigorous theoretical techniques. Informal models often provide more intuitive descriptions, but they also by their nature tend to be imprecise and/or incomplete. Formal models are very precise, but they are often considered obscure and difficult to use by non-experts.

[1]MOSTs and ArMOR do not form a memory model per se, as they cannot directly determine whether a proposed litmus test outcome is legal. However, they can be used to define components within models of the above types. As I tend to think in terms of multi-event axiomatic models, I tend to visualize MOSTs as being used in this way as well.

Formal hardware memory models can be classified into two main categories[2]. *Operational* models describe legal executions in terms of a series of steps taken by some hypothetical abstract machine. Each step represents a transition from one well-defined state to another according to some well-defined transition function. Operational models are often relatively intuitive, but they can suffer from a state space explosion that makes them less useful within practical tools [AMT14].

Axiomatic memory models define correct executions in terms of logical predicates that must be satisfied. One prominent type of predicate is the notion that a *happens-before graph*, or some subset thereof, be acyclic. A happens-before graph is a graph in which nodes represent instructions and/or events in some abstract machine and in which edges represent some formal notion that the source node happens prior to the destination node. A cycle of happens-before edges (i.e., an indication that an event happens before itself) is (in most cases) considered to be a proof by contradiction that the proposed scenario is impossible.

Axiomatic memory models based on happens-before graphs can be further subdivided into two categories [AMT14]. In *single-event axiomatic* models, each instruction in a program corresponds to one node in a happens-before graph. As Section 2.2.1 demonstrated, these models cannot naively check for cycles. They instead generally define legal executions as those for which certain carefully-defined subsets of the graph are acyclic and/or irreflexive. In *multi-event axiomatic* models, each instruction corresponds to multiple nodes in a happens-before graph. Multi-event axiomatic graphs are larger, but the analysis is often simpler than for single-event axiomatic models. Most often, any cycle in a multi-event axiomatic happens-before graph is enough to rule out an execution.

[2]Denotational models, which associate each state or program with a formal mathematical counterpart, form a third category. They are commonly used to define programming language semantics, but they are seldom used to define hardware memory models.

Neither axiomatic nor operational models have proven to be objectively superior, and both approaches are still in use today. However, the lack of a single standard approach often makes it difficult in general to compare any two models written using two different frameworks and languages. This in turn can make it difficult determine whether models are equivalent (e.g., the models of Figure 2.6 below), whether one model is stronger or weaker than another (Section 3.2), or how to map operations from one model to another [BMO+12, LGCP13, LPCZN13]. Even models which follow the same approach are often incompatible due to differences in the languages and formalisms used to specify them. Alglave et al. recently proposed the `herd` framework which defines a general-purpose langauge `cats` for defining single-copy atomic memory models; however, this only captures the subset of models which follow that approach. In response, Chapter 5 will define a general-purpose language for defining multi-event axiomatic memory models at the architecture and the microarchitecture levels.

In spite of the different approaches and languages used to define different memory models, researchers have had some success in manually proving equivalence between models [AMT14, MHMS+12, OSS09a]. This equivalence, once established, allows users and researchers to choose whichever form they prefer. Supposed proofs of equivalence have also at times been shown to be incorrect, and models thought to be equivalent have turned out to actually differ in subtle ways [AMT14, LPCZN13]. Nevertheless, the ability to prove equivalence between approaches remains a powerful mechanism for explaining and analyzing a wide variety of use cases for a wide variety of audiences.

2.2.3 Defining Models Which Reflect Actual Hardware

Ideally, a memory model would exactly match the behavior of real hardware implementations of that model: any behaviors observable on hardware would be considered legal by the model. In practice, hardware implementations may at times never pro-

duce behaviors which are permitted by the specification, and hardware may also permit behaviors which are forbidden by the specification. The latter is clearly erroneous, but the causes for this situation vary: there may be bug(s) in the model, in the hardware, or in both. Some hardware designs have been shown to incorrectly implement certain requirements of their architecture-level models [ABD+15, ARM11, Int15]. Researchers and designers also have proposed models which incorrectly forbade behavior that turned out to be observable on real hardware [AMT14]. Many memory model designs and implementations therefore end up following a pattern of repeated iteration and refinement, fixing discovered bugs at each step, with the goal of converging towards a properly-sound outcome.

The reasons for implementing stricter-than-necessary ordering enforcement can be subtle. Often, it may even be intentional. For example, a company may want to implement a simpler core design which is nevertheless fully software-compatible with more complex, higher-performance designs. Another reason is that a memory model may be specified in a way that allows for future optimizations that are not yet implemented in any hardware, as with load buffering on the Power architecture [SSA+11]. In cases like these, outcomes forbidden by the model must remain unobservable in hardware, but certain outcomes that are permitted by the model may never be observed on actual hardware.

As an example of the kinds of discrepancies that can arise, Figure 2.6 shows two litmus tests being analyzed using various formal and informal models of the Power architecture. The first test, `mp+lwsync+addr-po-detour`, is permitted by two formalizations but forbidden by a third, and it is unclear whether the industry documentation forbids or permits it. The second test, `mp+lwsync+addr-bigdetour-addr`, is similar, except that is forbidden by two of the three models. These two tests clearly demonstrate that the three formal models disagree, and that there is not yet a con-

Thread 0	Thread 1	Thread 2
`stw [x], 2`	`lwz r1, [y]`	`stw [x], 1`
`lwsync`	`xor r5, r1, r1`	
`stw [y], 1`	`lwz r2, [z+r5]`	
	`lwz r3, [x]`	
	`lwz r4, [x]`	
Proposed Outcome: `[x]=1, 1:r1=1, 1:r2=0, 1:r3=0, 1:r4=1`		

(a) Litmus test `mp+lwsync+addr-po-detour`. The naming convention is as follows: "`mp`" refers to the high-level pattern being tested (see Figure 2.2). `lwsync` refers to the synchronization used in thread 0. `addr-po-detour` refers to the synchronization used in thread 1: an address dependency, followed by a program order relationship, followed by a `detour` relationship. The latter appears to be a typo; `detour` is described as (`po` ∩ (`coe; rfe`)), while (`po` ∩ (`fre; rfe`)), which is used in this test, is labeled `rdw`, for "read different writes". To avoid confusion, I maintain the original name.

Thread 0	Thread 1	Thread 2
`stw [x], 1`	`lwz r1, [y]`	`stw [z], 2`
`lwsync`	`xor r5, r1, r1`	`lwsync`
`stw [y], 1`	`lwz r2, [z+r5]`	`stw [w], 1`
	`lwz r3, [w]`	
	`xor r6, r3, r3`	
	`lwz r4, [x+r6]`	
Proposed Outcome: `1:r1=1, 1:r2=0, 1:r3=1, 1:r4=0`		

(b) Litmus test `mp+lwsync+addr-bigdetour-addr`, where `bigdetour` refers to `detour`/`rdw` with extra synchronization added in thread 2.

Model	Style	`mp+lwsync+addr--po-detour`	`mp+lwsync+addr-bigdetour-addr`
[IBM13]	Informal	(unclear)	(unclear)
[SSA+11]	Operational	Forbidden	Forbidden
[MHMS+12]	Multi-Event Axiomatic	Permitted	Forbidden
[AMT14]	Single-Event Axiomatic	Permitted	Permitted
—	Real Hardware	Observed	Not Observed

(c) Disagreement of various models with hardware and with each other

Figure 2.6: The Power memory model still has some unresolved corner cases; there is not yet a consensus on the correct behavior for programs such as the two litmus tests shown here [AMT14].

sensus on the "officially correct" behavior in these corner cases, highlighting that increased formalism does not in itself solve the problem of completeness.

The discussion above motivates two key features in this thesis. First, it motivates the need to be able to rigorously compare an architectural memory model specification and a microarchitectural implementation of that specification. This comparison must also allow for the case in which an outcome is permitted by the architecture but not observable on a given microarchitecture. Second, it motivates the need for specification formats which are flexible and which can easily be adjusted when necessary. Both points are addressed throughout the rest of this thesis.

2.2.4 Other Subtle Ordering Relationships

This section describes various other properties related to orderings between instructions. All four are closely related and interdependent yet subtly different; memory models vary widely in whether (and how) they do or do not enforce these properties. As such, it is important that memory model analysis techniques (including those proposed in this thesis) be able to distinguish between them and reason about them.

Store Atomicity. Atomicity is heavily overloaded word. Its usage in the context of memory consistency models refers to the manner in which stores perform with respect to different cores in the system. A store is *single-copy atomic* if it performs with respect to every core in the system (including the issuing core) at a single point in time [Col92]. A store is *multi-copy atomic* if it performs with respect to every core other than the issuing core at a single point in time. Multi-copy atomic stores may perform with respect to the issuing core early. *Non-multi-copy-atomic* stores may perform with respect to cores in any order without restriction, although in general they still perform with respect to the issuing core first.

Store atomicity is beneficial because it ensures that transitivity and causality (discussed below) are automatically enforced, meaning that there is no need to deal

with the idiosyncrasies of cumulativity (also discussed below). It is the lack of multi-copy atomicity that allows many of the most counterintuitive behaviors to become visible. On the other hand, store atomicity allows fewer forms of buffering than does non-atomicity, and so performance may be degraded as compared to the optimum. Both choices are in widespread use; TSO is multi-copy atomic, but ARM and Power are not, for example.

Transitivity and Causality. Transitivity is the property that, for some events A, B, and C, if A happens before B and B happens before C, then A happens before C as well. Causality is the property that, for some events A and B, if A causes B to happen (or to happen in a certain way), then A should happen before B. Violations of causality and/or transitivity are generally considered highly counterintuitive to most programmers. Nevertheless, both can occur on some weakly-ordered architectures.

As an example of both properties, Figure 2.7 demonstrates how both transitivity and causality can be violated in practice. This example assumes that stores are not multi-copy atomic but that loads perform with respect to all cores in a single step. Because the proposed outcome of Figure 2.7a states that (i2) returns the value 1, (i2) must perform after (i1) has performed with respect to thread 1. Likewise, (i4) must perform after (i3) has performed with respect to thread 2. Due to the dependencies[3], (i3) may only perform after (i2) has performed, and (i5) may perform only after (i4) has performed.

By the above reasoning, since (i1) must happen before (i2) and (i2) must happen before (i3), it would appear that due to transitivity, (i1) must happen before (i3). Likewise, by causality, since (i1) created the value that (i3) wrote to memory, it would again appear that (i1) must happen before (i3). Unfortunately, neither intuition is guaranteed to hold true on weakly-ordered architectures such as Power and ARM. Figure 2.7b describes one execution which produces the outcome proposed in

[3]ARM and Power maintain the dependency ordering even though `r1 xor r1` is always zero.

Thread 0	Thread 1	Thread 2
(i1) `stw [x], 1`	(i2) `lwz r1, [x]`	(i4) `lwz r1, [y]`
	`xor r2, r1, r1`	`xor r2, r1, r1`
	(i3) `stw [y+r2], r1`	(i5) `lwz r3, [x+r2]`
Proposed Outcome: `1:r1=1, 2:r1=1, 2:r2=0`		

(a) Litmus test code

Thread 0	Thread 1	Thread 2
(i1) perf. wrt. thread 0		
(i1) perf. wrt. thread 1		
	(i2) perf. globally	
	(i3) perf. wrt. thread 1	
	(i3) perf. wrt. thread 2	
		(i4) perf. globally
		(i5) perf. globally
	(i3) perf. wrt. thread 0	
(i1) perf. wrt. thread 2		

(b) Timeline of one execution producing the proposed outcome

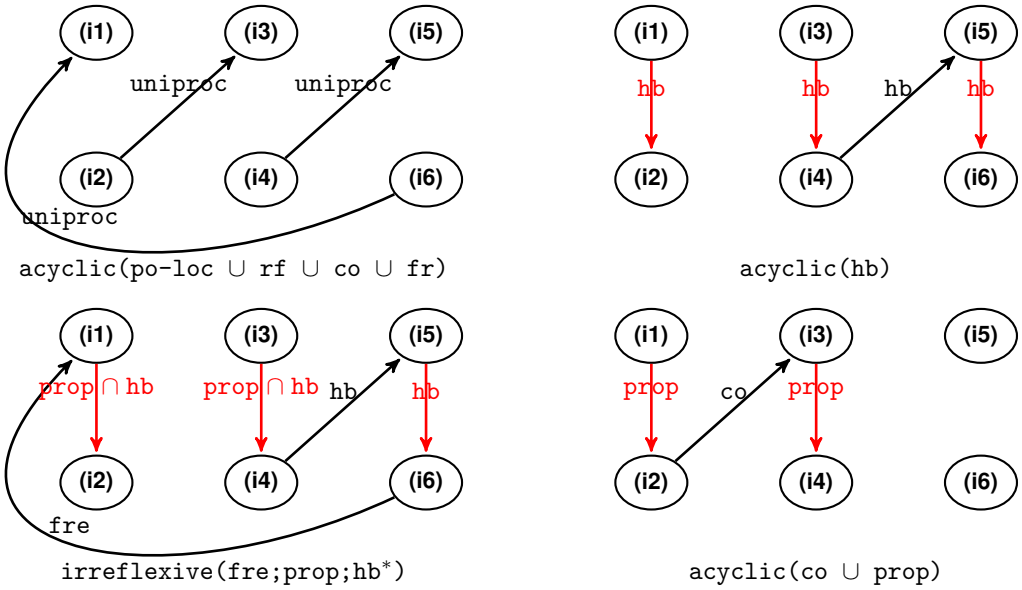
Figure 2.7: Litmus test `wrc+addrs`. The proposed outcome is permitted on Power, even though it violates causality and transitivity.

Figure 2.7a while satisfying all of the above constraints. The key distinction is that reasoning performed from the point of view of one core or thread need not hold true from the point of view of other threads. In particular, although (i1) must have performed with respect to thread 1 before (i2) performs, it does *not* have to perform with respect to thread 2 before (i2) performs. This breaks both transitivity and causality.

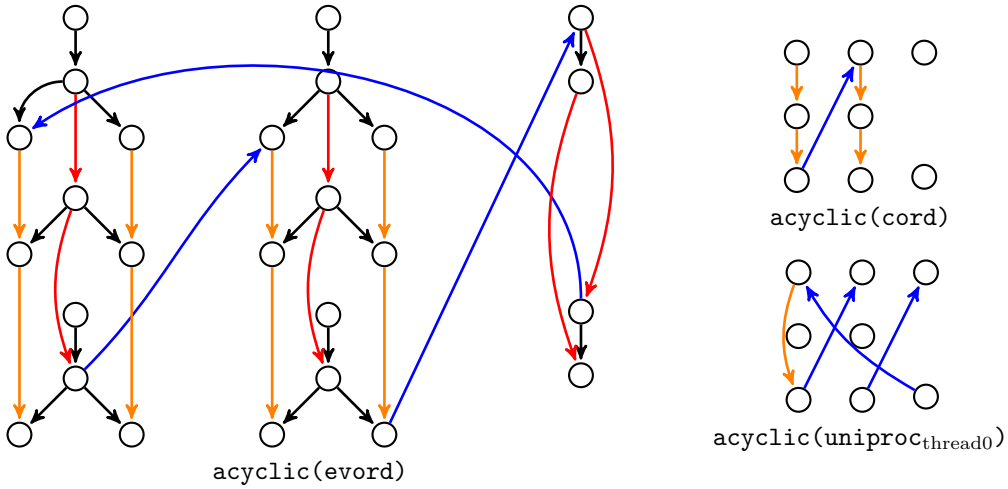
Figure 2.8 demonstrates how axiomatic models capture the lack of transitivity and causality for architectures such as Power. Figure 2.8a presents Power litmus test `blw-w-006`. This test highlights the fact that in the Power model, edges due to `lwsync` fences cannot be transitively composed with edges due to coherence orderings in order to form cycles that rule out an execution. In the single-event axiomatic approach of Figure 2.8b, the lack of transitivity is represented by searching for cycles (or reflexive

Thread 0	Thread 1	Thread 2
(i1) stw [x], 1	(i3) stw [y], 2	(i5) lwz [z], r1
lwsync	lwsync	xor r2, r1, r1
(i2) stw [y], 1	(i4) stw [z], 1	(i6) lwz [x+r2], r3
Proposed Outcome: 2:r1=1, 2:r3=0		

(a) Power litmus test `blw-w-006` [SSA+11]



(b) Single-event models [AMT14] check a number of subgraphs for acyclicity or irreflexivity. Although the union of the graphs would be cyclic, each individual relevant subgraph is acyclic/irreflexive.



(c) Multi-event models [MHMS+12] generally check a small number of graphs for cyclicity. The graphs themselves are bigger and more complicated, but the analysis (i.e., cycle checking) is simpler.

Figure 2.8: Comparing single-event and multi-event axiomatic models

Thread 0	Thread 1	Thread 2
(i) st [x]←1	(ii) ld [x]→r1	(v) ld [y]→r2
	(iii) sync	(vi) st [y]←3
	(iv) st [y]←2	

Outcome r1=1, r2=2:
Group A of (iii) = {(i),(ii)}
Group B of (iii) = {(iv),(v),(vi)}

Figure 2.9: Since Power's `sync` is A- and B-cumulative, it includes accesses from other threads into its scope. Most [ARM13, IBM13, SSA+11] but not all [AMSS10] formalizations consider (vi) to be in group B.

relations) in particular subsets of the graph rather than within the graph as a whole. In the multi-event axiomatic approach of Figure 2.8c, five separate graphs (three of which are pictured) are simply checked for cycles; no subsets need to be taken.

Cumulativity. Power and ARM hardware enforce transitivity and causality only when *cumulative* fences are inserted. Cumulativity is a recursively defined property. The base case is that instructions program-order-before a given fence are members of "group A", and instructions program-order-after a fence are members of "group B". With A-cumulativity, instructions *from any thread* which happen before a member of group A are themselves included in group A, recursively. Likewise, with B-cumulativity, instructions *from any thread* which happen after a load from that thread returns a value written by a store in group B are themselves members of group B, recursively. A cumulative fence is defined to enforce ordering between each member of group A and each member of group B matching the specification of the fence type[4].

Figure 2.9 demonstrates the cumulativity of the Power `sync` fence (iii). In the base case, group A consists of (ii) and group B consists of (iv). Then, since (ii) reads from (i), (i) happens before (ii), and so since the `sync` is A-cumulative, (i) is included into group A of the `sync` instruction. Similarly, (v) reads from (iv), and (vi) happens after (v), so (v) and (vi) are included in group B of the `sync` by B-cumulativity.

[4]e.g., the Power `lwsync` fence is cumulative but does not enforce ordering between stores in group A and loads in group B.

Cumulativity provides a compromise between the performance benefits of allowing transitivity and causality violations, while allowing for the ability to restore both properties where necessary. Unfortunately, because cumulativity is inherently a dynamic relationship, it can be hard to reason about the orderings it implies, and many models of ARM and Power require groups A and B to be calculated using some explicit form of iteration until convergence. Likewise, as "happens before" can be understood in many ways, the use of "happens before" within the definition of cumulativity has been formalized in many competing ways [Alg12, AMT14, ARM13, IBM13, MHMS$^+$12, SSA$^+$11].

2.3 Empirical Analysis of Memory Models

Memory models can be evaluated in a number of ways. First and foremost is correctness: implementations (software, microarchitecture, or otherwise) should always meet whatever correctness specifications are required. Once correctness is established, performance can be measured. Performance is hard to quantify exactly, and it is specific to an implementation rather than a specification. A third important metric is ease of use, which is much harder to quantify, but nevertheless very important in practice.

This thesis takes the point of view that correctness should be established before performance results can be considered meaningful. Very often, microarchitectures may be able to realize large performance gains directly from even subtle violations of correctness. In other words, the cost of maintaining 100% correctness may be large as compared to maintaining even "99% correctness" [BA08, LPCZN13]. If correctness has not yet been established for a new architectural proposal, then it can be unclear whether the performance gains are due to a true contribution of the proposal or simply due to an as-yet-undiscovered bug. A goal of all of the techniques and tools

developed in this thesis is to make a verification-first approach more practical than is often feasible with the tools and resources available today.

2.3.1 Correctness

Formal proofs of correctness for any given implementation can be prone to unexpected bugs or corner cases. The formalism may assume some property which is not in fact true, meaning that any result derived from the faulty axiom itself becomes faulty. Or, the implementation may not actually enforce the rules of the specification correctly. Knuth famously summed up this problem in the following quote: "Beware of bugs in the above code; I have only proved it correct, not tried it." [Knu97].

A standard approach for empirically verifying the correctness of an implementation is testing: executing well-defined test cases with known expected outcomes, and then seeing if the observed outcome matches the expected outcome. Testing-based techniques are generally much faster than formal verification, and they can be used as a sanity check that no assumptions made during formalization are violated in practice. However, testing techniques generally suffer from a lack of coverage that formal models can often better provide, as many bugs may be triggered only in extremely uncommon corner cases. For this reason, formal verification and empirical testing are generally used to complement each other.

Testing-based techniques have proven valuable for performing empirical correctness checking. Hangal et al. developed TSOTool, a framework for automatically generating tests for the TSO memory model and then for using the test results to track down the hardware origin(s) of any bugs [HVML04]. Many others have developed similar frameworks for other use cases [MHC+06]. Researchers have also proposed architectural structures for dynamically watching for ordering violations [CL02, CLH+09, CMP08, MS05].

Testing often does identify bugs in hardware, but like everything else it comes with tradeoffs. Early (e.g., pre-silicon) testing may be slower and/or incomplete, while later (e.g., post-silicon) testing may be more comprehensive due to the ability to test real hardware. However, post-silicon bugs are dramatically more expensive to correct due to the time and money cost of fabricating a new corrected version. Both pre- and post-silicon approaches have become standard as the burden of verification has grown increasingly difficult over time.

2.3.2 Performance

The key motivation for defining and using weak memory models is the performance benefits they enable. However, it is a slight misnomer to talk about the performance of a given memory model itself, as a model is just a specification. It is more correct to talk about the ability of a memory model to allow (or not allow) for high-performance implementation(s). Furthermore, as with any studies of performance, the choice of benchmarks can make a big difference in the final scores for each case.

The extent to which the choice of memory model affects the ability to build high-performance implementations is unclear. Researchers have proposed various hardware optimizations using techniques such as speculation in order to build microarchitectures for models as strict as SC yet which apparently have low to negligible performance overhead as compared to weaker models [BMW09, CTMT07, DMT13, HS13, SNM+12]. However, industry does not appear to have reached the same conclusions. In any case, the power, area, and/or verification costs of building such proposals in real hardware may limit their applicability.

Many features of weak hardware and software memory models exist so that "expert" programmers can take advantage of them when writing key libraries or data structures [BA08]. Often, these features are added with particular benchmarks in mind. One prominent example is the C++11 `memory_order_consume` parameter,

which behaves similarly to an acquire operation except that it only enforces orderings for dependent instructions. This feature was added to C++ in large part so that a Read-Copy-Update (RCU) data structure could be efficiently and portably implemented within the Linux kernel [MRP+14].

An additional concern is the ability to write high-performance software and compilers that can take full advantage of the available hardware. Lê et al. built, proved correct, and measured the performance of a work-stealing queue and a FIFO data structure in C11, ARM, and Power [LGCP13, LPCZN13]. They showed that performance can vary dramatically across memory models, and that sometimes, the optimal design for one memory model is sub-optimal on others[5]. This sequence of papers demonstrates the need for better and more reliable toolchains when putting memory model analysis techniques into practice.

2.4 Comparing and Mapping Between Models

A key practical challenge for consistency model specification and analysis techniques is to enable synchronization primitives to be correctly mapped from one model to another. Although programmers generally work at the top of the stack of models shown in Figure 2.1, the code must be mapped onto the lower levels of the stack, generally through one ore more intermediate layers (e.g., compiler intermediate representations), before it can be executed by hardware. Unfortunately, as Section 2.2.2 showed, the frameworks used to define different memory models may in general differ widely in the approach used (operational vs. axiomatic vs. informal) and the languages used to define them. As a result, it can become very difficult to reason about how to map consistency model features between models.

[5]Highlighting the difficulty of endeavors such as these, the analysis was later shown to itself be slightly buggy [ND13]. Their conclusion remains valid nevertheless.

	memory_ order_ relaxed	memory_ order_ release	memory_ order_ seq_cst
x86	mov	mov	xchg
Power	st	lwsync; st	sync; st
ARMv7	st	dmb; st	dmb; st
ARMv8	str	stl	stl
Itanium	st.rel	st.rel	st.rel; mf

Table 2.1: Mapping C11 low-level atomics with different ordering specifications onto hardware. The software constructs map onto different architectures in different ways [S+].

As just one example, consider the process of compiling C11 atomic stores onto different architectures. C11 is a data-race-free language, meaning that all cross-thread communication must take place via explicit synchronization operations [AH90, ISO11b]. In C11 terminology, two types of operations are considered synchronization operations: operations on mutexes, and loads and stores which are annotated as being "atomic"[6]. Atomic operations in turn come with a `memory_order` parameter indicating the ordering semantics associated with that instruction. Various types are available, including sequentially consistent semantics, acquire/release semantics [GLL+90], and relaxed semantics, where the latter implies indivisibility but no inter-instruction ordering requirements. Use of `memory_order` parameters other than `memory_order_seq_cst` is described by the designers as being intended for experts only [BA08]

Table 2.1 shows how three flavors of C11 atomic store orderings map onto different architectures in very different ways [S+]. The `memory_order_release` and `memory_order_seq_cst` options map onto some combination of normal memory access opcodes (e.g., `mov`), opcodes which combine memory access and synchronization (`stl`, `st.rel`), and normal opcodes plus explicit fences (`lwsync; st`, `dmb; st`). No one type of mapping is universal. The corresponding table for loads is even more

[6] "Atomic" is a heavily overloaded word; it it used here 1) to indicate an indivisible sequence of operations, and 2) to annotate memory accesses as being synchronization operations. It does not imply store atomicity as defined in Section 2.2.4.

diverse, enforcing orderings through the use of features such as explicit false dependencies. The process of deriving these mappings and proving them correct is generally shielded from application programmers, but they represent very real complexity for library and compiler writers.

Unfortunately, the current most reliable method for determining such mappings requires the construction of complicated formal models and dense mathematical correctness proofs which may take years to complete [BMO+12, Gha95, PVJ15]. Furthermore, the process of deriving mappings for new primitives can uncover unexpected and unintended incompatibilities between models. For example, Lê et al. discovered that a subtle incompatibility between the C11 and the Power memory models led to "unrecoverable overheads" in the design of work-stealing queues for Power [LPCZN13]. This problem will only continue to get worse as new languages and new paradigms introduce even more new features which have not yet been throughly formally analyzed [OCY+15, WBDB15].

Lastly, the goal of mapping primitives from one model to another carries an implicit assumption that doing so is even possible in the first place. Most models can be made to enforce even sequential consistency given a sufficient amount of synchronization, but there are exceptions. For example, on Itanium, although acquire loads and release stores can be made to behave in a sequentially consistent manner, unordered loads and stores cannot be made sequentially consistent through any amount of synchronization [Int10]. Furthermore, it is not clear whether many GPU memory models can be made to behave in a sequentially consistent manner through any choice of operations or any amount of synchronization [ABD+15, NVI13b]. Omissions such as these make it effectively impossible to map certain types of synchronization onto certain types of hardware.

2.5 Simplifying Abstractions Made in This Thesis

Many memory model specifications are somewhat abstracted from the details of any actual instruction set. Issues such as false sharing of cache lines, partially-overlapping accesses, different cacheability modes, ordering of intra-SIMD instruction accesses, etc., are often left unspecified by the formalism. Sometimes, memory models may abstract away hardware details because even the abstracted model may be difficult enough to build and/or analyze. In this sense, understanding the simplified model becomes a prerequisite for building an extended version which incorporates the finer details. Other times, the model may be intentionally abstracted so that the same model can be applied to different instruction sets. For example, the total store ordering (TSO) memory model was originally defined for SPARC and hence used `SWAP` (atomic swap) and `LDSTUB` (atomic test and set) as its synchronization primitives [SPA94a]. TSO has since been adapted to x86, which uses `mfence` and the LOCK prefix as its primary synchronization primitives. The spirit of both model instances is the same, but due to the difference in synchronization primitives, the implementations differ slightly.

Unless otherwise stated, in keeping with the standard approach taken in memory model research, this thesis makes the following basic assumptions about memory access behavior [AMT14, LPM14, MHMS+12, OSS09a, SSA+11]. First, it assumes that there are some set of basic memory access types: loads, stores, and synchronization accesses of various kinds (e.g., atomic read-modify-write). It does not distinguish between different types of stores, stores to different memory types (e.g., write-back vs. write-through vs. non-cacheable), etc. I also assume the existence of synchronization primitives in the form of fences, dependencies, the x86 LOCK prefix, and so on. However, the details of, e.g., the instruction sequences giving rise to particular types of dependencies are not generally elaborated upon in an effort to keep the focus on the memory model itself. Second, it assumes that values are identically-sized (at

the granularity at which cores address memory). Lastly, it assumes that accesses are non-overlapping, meaning that any two accesses either alias entirely (i.e., they are in fact accessing the same address) or not at all.

2.6 Chapter Summary

This chapter presented the necessary background for understanding the goals and mechanisms presented in this thesis. The subsequent chapters explore solutions to many of the problems posed by this chapter; namely, the need for clear and precise specifications of memory ordering behavior, the need for rigorous and systematic memory model analysis techniques, and the need to understand how memory models are implemented at the microarchitecture level.

Chapter 3

Memory Ordering Specification Tables and ArMOR

The previous chapter highlighted, among other things, the problem that memory models are often specified in ways that make it difficult to compare models or to map primitives from one model to another. This chapter introduces the Memory Ordering Specification Table (MOST), a architecture-independent format for defining memory model primitives [LTPM15a, Lus15]. It also introduces the ArMOR framework for comparing and manipulating MOSTs. The next chapter then uses the ArMOR framework for an in-depth case study of MCM-aware dynamic binary translation.

3.1 Introduction

Across both academia and industry, a large number of memory models have been studied in detail, formalized into rigorous definitions, and/or put into practice in the real world. As Chapter 2 showed, these memory models vary widely in relative strength, degree of formality, definition approach taken (i.e., operational, axiomatic, or other), choices of fence types to include, and so on. Furthermore, given that many of these specifications are vague, obscure, and/or simply incompatible with each

other, it can be challenging to check whether a given pair of ordering mechanisms from different models is equivalent or whether they differ in subtle ways. Such a need would arise when compiling ordering mechanisms from a given software model onto a given hardware model, to give just one example.

Due to the complexity of even properly defining memory models such as those used by C/C++11, Java, ARM, or Power, and due to the one-off manner in which memory models are often defined, any analysis or toolflow (e.g., any compiler or any synchronization library) can only be built in a reliably correct manner through the use of formal analysis [Alg12, BMO+12, BA08, LGCP13, LPCZN13, MHMS+12, MPA05, SSA+11]. Informal analyses are prone to being either overconstrained or simply incorrect. Unfortunately, the lack of any common specification language means that attempts to build formal cross-model analyses can take multiple person-years of dense formalism to complete.

This chapter presents ARchitecture-independent Memory Ordering Requirements (ArMOR), a model-independent framework for specifying, reasoning about, and translating between memory consistency models. ArMOR defines memory ordering requirements (MORs)—fences, dependencies, or any other ordering enforcement mechanisms—in a self-contained, complete, and precise format known as a Memory Ordering Specification Table (MOST). MOSTs resemble widely-used reordering tables (e.g., for TSO, as in Figure 2.3) that indicate whether load→load, load→store, store→load, and store→store orderings need to be maintained. However, a key contribution of MOSTs is that they also directly encode subtle details such as store multi-copy atomicity, fence cumulativity, and so on. This added precision makes MOST-based analysis less prone to the types of under- or over-constraints that can result from relying on less systematic techniques.

We envision MOSTs being used to specify the behaviors of memory models at the hardware, software, and/or intermediate representation level. While this thesis fo-

cuses on analyzing hardware models, approaches which are broadly similar to MOSTs have been considered at the software level as well [PVJ15]. In particular, this chapter focuses on the definition and analysis of MOSTs themselves, and it concludes with a gallery of MOSTs used to define a large number of widely-used hardware models. Chapter 4 then presents a series of in-depth case studies in which MOSTs are used to perform cross-instruction set dynamic binary translation.

The rest of this chapter is arranged as follows. Section 3.2 starts with a motivating example. Section 3.3 describes the MOST specification syntax. Section 3.4 presents the ArMOR framework for analyzing and manipulating and comparing MOSTs. Section 3.5 describes related work, and Section 3.6 concludes. Finally, Appendix A presents the MOST definitions for a range of popular hardware memory models [Lus15].

3.2 Motivating Example

Although many programmers write parallel code under the assumption of sequential consistency (SC), few software or hardware models today directly implement SC due to its performance cost. As a result, application programmers or library writers must explicitly specify additional consistency-related synchronization points, whether at coarse grain (e.g., function call or GPGPU kernel boundaries), medium grain (e.g., mutex operations, the Java `synchronized` keyword), or fine grain (e.g., C11/C++11 low-level atomics, inline assembly). As discussed in Section 2.4, one key challenge in each case is determining how to map a given software synchronization primitive onto a sufficiently strong hardware primitive in the target architecture.

Figure 3.1 highlights how dangerous memory model analysis pitfalls can arise from the use of informal specification languages and the lack of ability to easily compare features of different memory models. Figure 3.1 depicts a commonly-used manner of

TSO PPO

	Loads	Stores
Loads	✓	✓
Stores	—	✓
Stores are multi-copy atomic		

Power `lwsync`

	Loads	Stores
Loads	✓	✓
Stores	—	✓
A- and B-cumulative		

Processor Consistency

	Loads	Stores
Loads	✓	✓
Stores	—	✓
Stores are not multi-copy atomic		

IBM 370/390/z-Series

	Loads	Stores
Loads	✓	✓
Stores	—	✓
Store buffer forwarding forbidden		

Figure 3.1: Reordering tables for various architectures. All four appear deceivingly similar; they differ only in the subtle details encoded below the tables using natural language.

describing the ordering enforcement of various architectures and architectural mechanisms. Shown first is the total store ordering (TSO) consistency model used by SPARC and x86 [SPA94b, OSS09a]. This table was first presented in Section 2.2.1. More specifically, the figure depicts TSO *preserved program order* (PPO)—the subset of the original (per-thread) program order relationships that an architecture guarantees to maintain. As in Figure 2.3, the table in Figure 3.1 specifies whether an access of one type (the row heading) may be reordered with a program-order-subsequent access of another type (the column heading). Under TSO, stores may be reordered with later loads, but all other orderings are required. The table also specifies that on x86, stores are multi-copy atomic, as defined in Section 2.2.4.

Figure 3.1 also depicts the definition of three other architectures and/or features which appear to enforce very similar orderings to TSO. The figure depicts PPO for two other memory models: processor consistency and the IBM 370 memory model [GLL+90, IBM83]. It also depicts the orderings enforced by `lwsync`, a fence on the Power architecture. All three are loosely similar to TSO PPO in that they permit only store→load reordering. Nevertheless, they differ in the extent to which they enforce atomicity and/or cumulativity (Section 2.2.4), meaning that their apparent similarity is in fact misleading.

`iriw` litmus test in abstract form

Thread 0	Thread 1	Thread 2	Thread 3
(i1) st [x]←1	(i2) ld [x]→r1 (i3) ld [y]→r2	(i4) ld [y]→r3 (i5) ld [x]→r4	(i5) st [y]←1

Outcome r1=1, r2=0, r3=1, r4=0

`iriw` litmus test on x86

Thread 0	Thread 1	Thread 2	Thread 3
mov [x], 1	mov rax, [x] mov rbx, [y]	mov rcx, [y] mov rdx, [x]	mov [y], 1

Outcome rax=1, rbx=0, rcx=1, rdx=0 forbidden

`iriw+lwsyncs` litmus test on Power

Thread 0	Thread 1	Thread 2	Thread 3

stw [x], 1	lwz r1, [x]
	lwsync
	lwz r2, [y]
		lwz r3, [y]
		lwsync
		lwz r4, [x]
			stw [y], 1

Outcome r1=1, r2=0, r3=1, r4=0 observable

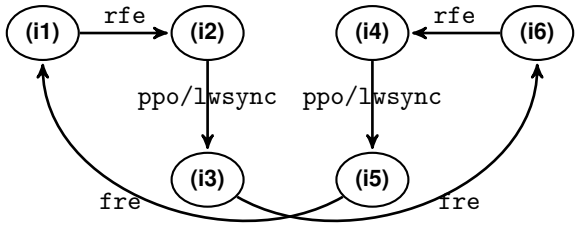


Figure 3.2: The `iriw` litmus test in various forms. The proposed outcome is observable on Power (with `lwsync` fences) but not on x86-TSO, indicating that the orderings enforced by TSO PPO and by `lwsync` differ. Even experts have been prone to the pitfall of assuming that similarities between tables in Figure 3.1 indicate equivalence of the features being represented [SHW11].

Subtle differences between the tables in Figure 3.1 can lead to pitfalls when overly-informal analyses are used. For example, as a sneak preview of Chapter 4, consider the problem of executing TSO code on a processor implementing the Power architecture memory model [ŠVZN+13]. Memory accesses on Power may be reordered liberally by default; orderings on Power are only enforced through inter-instruction dependencies or explicit fences such as `lwsync`. Given the tables in Figure 3.1, it may appear that insertion of `lwsync` between every pair of accesses should be sufficient to restore all of the orderings required by TSO. However, this appearance is deceiving, as the two are in fact *not* equivalent, and `lwsync` is *not* sufficient to restore TSO. The difference in strength between the default orderings of TSO and the orderings enforced by `lwsync` can be demonstrated explicitly by a litmus test called `iriw` (independent reads of independent writes), shown in Figure 3.2. In particular, although TSO enforces orderings between the Thread 0 store to `[x]` and the Thread 1 load of `[y]` and between the Thread 3 store to `[y]` and the Thread 2 load of `[x]`, `lwsync` on Power does not.

This example shows that subtle or seemingly-unimportant low-level details of a memory model can result in differences which are visible to the programmer and which therefore must be properly documented. If the language used to specify orderings states only that load→load, load→store, and store→store orderings need to be enforced, but does not specify the degree of atomicity and/or cumulativity, then the specification is incomplete at best and prone to unexpected or buggy outcomes at worst.

This chapter avoids the pitfall of the above example by encoding information about atomicity and cumulativity directly within the reordering tables themselves. This added precision is what distinguishes MOSTs from the more common and less precise tables shown in the figure. Each cell in a MOST lists not just an ordering, but also the *strength* of the ordering (i.e., whether it is single-copy atomic, multi-

copy atomic, or neither). New rows and columns are introduced to directly address cumulativity (as required by `iriw` above). The details of these new features are elaborated in Section 3.3. The highlight, however, is that the MOSTs will be able to clearly distinguish between the models in a way that is both clear to humans and amenable to algorithmic analysis.

3.3 Memory Ordering Specification Tables

MOSTs describe the reordering behavior of memory consistency models at a precise and detailed level sufficient to support algorithmic analysis and automated comparisons and translation. Just as with traditional reordering tables, each cell in a MOST specifies whether instructions of the type in the row heading must maintain their ordering with subsequent instructions of the type in the column heading. Traditional reordering tables are most often used to define preserved program order, the set of program order relationships which are maintained by default. In contrast, MOSTs can be used to define not just preserved program order, but also fences or any other type of ordering enforcement mechanism, as shown with `lwsync` in the previous section. As will be discussed below, where necessary, MOSTs also include orderings that are enforced between accesses of different threads.

As two running examples, we will derive the MOSTs for TSO preserved program order and for Power `lwsync` step by step. The complete MOSTs will be given at the end of this section once all of the necessary notation and details have been presented.

3.3.1 Store Atomicity and Ordering Strength

The first source of imprecision in traditional reordering tables is the fact that they do not address how orderings may have different *strengths*. In particular, stores may in general *perform* with respect to (i.e., become visible to) different cores in a system

Thread 0	Thread 1	Thread 2
st [x]←1	ld [x]→r1	ld [y]→r2
	fence	fence
	st [y]←1	ld [x]←r3

Outcome r1=1, r2=1, r3=0:
Forbidden if stores are single-or multi-copy atomic
Allowed if stores are non-multi-copy-atomic

Figure 3.3: The `wrc+ncfences` litmus test with non-cumulative fences. Note the subtle distinction from Figure 2.7.

at different times. The restrictions on such visibility events can be summarized into three degrees of store atomicity, as described in Section 2.2.4 and recapped below. *Single-copy atomic* stores must become visible to all cores in the system at a single time [AM06]. Single-copy atomicity is uncommon, as it forbids even forwarding from a private local store buffer. *Multi-copy atomic* stores must become visible to all cores besides the issuing core simultaneously. In other words, a multi-copy atomic store cannot ever be visible to some but not all remote cores. TSO (used by SPARC and x86) falls into this category. *Non-multi-copy-atomic* stores may become visible with respect to remote cores in any order and in any number of steps. Power and ARM fall into this category.

The effect of store atomicity (or a lack thereof) is commonly depicted by the write-to-read causality litmus test (`wrc+ncfences`) of Figure 3.3. This test works as follows. If the thread 1 load reads the value written by the thread 0 store and then forwards it along to thread 2, must thread 2 have also seen the effect of the thread 0 store? If the store from thread 0 is multiple- or single-copy atomic, then thread 2 must see the thread 0 store before it sees the thread 1 store. However, if the store from thread 0 is not multi-copy atomic, then the thread 1 store may propagate to thread 2 *before* the thread 0 store does, *even though the thread 0 store executed first*. This violates the intuitive notion of causality: even though the thread 0 store *causes* the thread 1 store value to exist, the thread 0 store need not become visible to

Abb.	Description
$\checkmark_S$	Single-copy atomic
$\checkmark_M$	Multi-copy atomic
$\checkmark_N$	Not multi-copy atomic
—	Unordered

(a) Store→store

Abb.	Description
✓	Ordered
$\checkmark_L$	Locally ordered
—	Unordered

(b) Store→load

Abb.	Description
✓	Ordered
—	Unordered

(c) Other

Figure 3.4: MOST strength levels used in this thesis

	Ld	St
Ld	✓	✓
St	—	$\checkmark_M$

(a) TSO

	Ld	St
Ld	✓	✓
St	—	$\checkmark_S$

(b) IBM 370/390/zSeries

Figure 3.5: The addition of explicit strength levels allows MOSTs to distinguish cases that would appear identical using traditional reordering tables. These MOSTs include only properties addressed up through and including Section 3.3.1; the full MOSTs are shown in Figure 3.7.

other threads before the thread 1 store. Nevertheless, this execution remains a legal outcome on non-multi-copy-atomic architectures such as ARM and Power.

To account for such strength differences in an architecture-independent manner, we introduce various *strength levels* into our MOST notation. Figure 3.4 summarizes the ordering strength levels used to describe MORs for architectures surveyed in this thesis. Additional (e.g., scoped) strength levels could easily be added if necessary. As an example of the benefit of these strength levels, Figure 3.5 shows MOSTs for the TSO and IBM 370/390/zSeries memory models. With traditional reordering tables, the architectures would appear equivalent. With the improved precision of MOSTs, the difference in store→store ordering strength is made explicit.

3.3.2 Same-Address Dependencies

Accesses from the same thread to the same address generally must maintain the ordering specified by program order. This property is sometimes called *coherence*[1].

[1]As discussed in Section 2.1.3, coherence protocols often use stronger definitions of coherence (e.g., single writer or multiple readers), while other consistency model papers may use weaker notions such as total orders only on stores to the same address. Chapter 6 explores this question in greater detail.

Thread 0	Thread 1	Thread 2
(i1) st [x]←1	(i4) st [y]←1	(i5) ld [y]→r3
(i2) ld [x]→r1		(i6) ld [x]→r4
(i3) ld [y]→r2		
Outcome: r1=r3=1, r2=r4=0: Allowed		

Figure 3.6: TSO litmus test n7 [OSS09a]. Although (i1) and (i2) access the same address, that store→load same-address ordering is *not* enforced from the point of view of other observers.

There are exceptions; SPARC RMO and old Power models relax load→load orderings to the same address, while the behavior is forbidden yet observable on some GPUs [ABD+15, AMT14, SPA94b, TDF+02]. To address this in MOSTs, we explicitly distinguish accesses to the same address (labeled in the MOSTs in this thesis as "SA") from those to different addresses ("DA").

The notion of ordering strength from the previous subsection is also relevant to per-address orderings. In particular, a store→load ordering may need to be enforced locally to ensure that each load returns the value written by the latest store to the same address. However, the same store→load ordering may *not* need to be enforced from the point of view of any remote observers. This somewhat surprising example is highlighted in Figure 3.6. In this example, (i6) can occur after (i1) but before (i2) becomes visible to thread 2. In other words, from the point of view of thread 2, the (i1) happens after the (i2), even though (i1) and (i2) access the same address. This re-emphasizes the need not just to specify that orderings must be enforced, but also to precisely specify their strength.

This added detail is enough to complete the MOSTs for SC, IBM 370, TSO, and RMO PPO, as shown in Figures 3.7b and 3.7c, respectively. In particular, for TSO, store→store ordering has been marked as being multi-copy atomic, and store→load ordering is marked as being enforced, but only locally, if the instructions access the same address. Figure 3.7d also shows how the MOST for SPARC RMO clearly

	Load	Store
Load	✓	✓
Store	$\checkmark_S$	$\checkmark_S$

(a) Sequential Consistency PPO

	Load to Same Address	Load to Diff. Address	Store
Load	✓	✓	✓
Store	$\checkmark_S$	—	$\checkmark_S$

(b) IBM 370/390/z-Series PPO

	Load to Same Address	Load to Diff. Address	Store
Load	✓	✓	✓
Store	$\checkmark_L$	—	$\checkmark_M$

(c) TSO PPO

	Load to Same Address	Load to Diff. Address	Store to Same Address	Store to Diff. Address
Load	—	—	✓	—
Store	$\checkmark_L$	—	$\checkmark_M$	—

(d) RMO PPO

Figure 3.7: Complete MOSTs for various single- or multi-copy atomic models

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO St	BC St
PO Ld	✓	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓	✓
PO St	—	—	—	$\checkmark_N$	$\checkmark_N$
AC St	—	—	—	$\checkmark_N$	$\checkmark_N$

(a) Power `lwsync`

	PO Ld	BC Ld	PO St	BC St
PO Ld	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓
PO St	✓	✓	$\checkmark_S$	$\checkmark_S$
AC St	✓	✓	$\checkmark_S$	$\checkmark_S$

(b) Power `sync`

Figure 3.8: Incorporating cumulativity into MOST definitions

indicates that load→load ordering of accesses to the same address does not need to be enforced.

3.3.3 Fence Cumulativity

Another property of MORs on many weakly-ordered architectures is cumulativity (Section 2.2.4). MOSTs address cumulativity by including A-cumulative (AC) and B-cumulative (BC) operations as explicit rows and columns. Orderings of accesses related by cumulativity are specified in MOSTs in exactly the same way as for accesses

related by program order or in the same thread as the MOR in question. Figure 3.8 shows the MOSTs for the `lwsync` and `sync` fences of the Power architecture. The fact that the `sync` fence (iii) enforced ordering from (i) to (iv) in Figure 2.9, for example, is captured by the $\checkmark_N$ entry in row (AC St) and column (PO St). From the point of view of (iii), (i) is related by A-cumulativity, and (iv) is later in program order.

3.3.4 Summary

By incorporating the details discussed above, MOSTs serve as a precise, architecture-independent, and self-contained specification of the semantics of memory ordering requirements (MORs). To demonstrate the usefulness of this approach, the next section describes how to algorithmically compare and manipulate MOSTs. In addition, Appendix A includes a comprehensive gallery of MOSTs for numerous architectures and architectural features [LTPM15b].

3.4 ArMOR: Comparing and Manipulating MOSTs

A key benefit of the MOST notation is that it allows for flexible, algorithmic comparison of MOSTs, even when they originally come from different architectures. This flexible comparison forms a key component of the compiler, mapper, and translator use cases envisioned earlier in Section 3.1. Taken together, the comparison and manipulation approaches described in this chapter form the *Architecture-Independent Memory Ordering Requirements* (ArMOR) framework.

3.4.1 MOST Partition Refinement

Because different architectures emphasize different consistency model features, as described in Section 3.3, they may use distinct choices of rows and columns to define their MOSTs. To resolve this, before any MOST-MOST comparisons can occur, the

TSO PPO Pre-Refinement

	PO+ SA Ld	PO+ DA Ld	PO St
PO Ld	✓	✓	✓
PO St	$\checkmark_L$	—	$\checkmark_M$

Refine →

TSO PPO Mid-Refinement

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO St	BC St
PO Ld	✓	✓	*?*	✓	*?*
AC Ld	*?*	*?*	*?*	*?*	*?*
PO St	$\checkmark_L$	—	*?*	$\checkmark_M$	*?*
AC St	*?*	*?*	*?*	*?*	*?*

(a) Because cumulativity is not explicitly addressed by the TSO PPO specification, the MOST must be refined in order to compare it with MOSTs from the Power architecture. However, not every cell in the refined MOST can be determined from the original MOST.

TSO PPO Post-Refinement

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO St	BC St
PO Ld	✓	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓	✓
PO St	$\checkmark_L$	—	✓	$\checkmark_M$	$\checkmark_M$
AC St	✓	✓	✓	$\checkmark_M$	$\checkmark_M$

(b) The entries with unknown values can be filled in by reasoning that the multi-copy atomicity of TSO directly implies that cumulativity holds automatically.

Figure 3.9: Using MOST partition refinement to compare TSO PPO and Power `lwsync`

rows and the columns of the MOSTs must be *refined* into matching partitions. The MOST refinement process has two steps. The first is to find the set of categories that should be used as the row and/or the column headings for the refined MOSTs. Standard partition refinement techniques can be used to merge the row and/or column choices from different MOSTs into a finer-grained partition capturing both; thus we omit a full algorithmic description here [PT87].

The second step is to fill in the cells of the newly-refined MOST. In most cases, this simply requires duplicating the original contents of a cell that was refined into multiple "child" cells. However, if a particular MOST feature is architecture-specific, partition refinement can lead to scenarios in which the ordering strength of a particular cell is left unspecified. These cells can be filled in conservatively (i.e., by assuming the unspecified orderings are required, or by assuming they are not enforced) or using some external reasoning.

Figure 3.9 shows an example. The MOST for the `lwsync` fence of the Power architecture (Figure 3.8) is laid out differently from the MOST defining TSO PPO (Figure 3.7c), as TSO does not explicitly define its MOSTs in terms of cumulativity. Here, the two MOSTs cannot yet be compared directly, even though Section 3.2 motivated the need for performing this comparison rigorously. Refining the partitions of TSO PPO to match `lwsync` produces the MOST on the right side of Figure 3.9a, and this MOST has cells whose entries cannot be directly determined from the contents of the original TSO PPO MOST. In this particular case, we can reason that cumulativity follows implicitly from the multi-copy atomicity of TSO, and therefore the cumulative ordering cells are in fact enforced. We therefore fill in the cells corresponding to cumulative orderings in the manner implied by multi-copy atomicity, resulting in the refined MOST of Figure 3.9b.

3.4.2 ArMOR Comparison Operators

Once two MOSTs have been refined (if necessary) into the same layout of rows and columns, then a comparison of the two can be defined by comparing each pair of corresponding cells. The cell-by-cell comparison is defined by checking whether one strength level implies the other. For example, enforcement of single-copy atomic store→store ordering of strength $\checkmark_S$ implies that multi-copy atomic store→store ordering of strength $\checkmark_M$ is also enforced, and hence that $\checkmark_S \geq \checkmark_M$. We define the full complement of comparison operations ($<$,$\leq$,$=$,$\neq$,$\geq$,$>$) analogously. Note that in general, this ordering is partial, not total.

Two MOSTs may also be combined to produce a single MOST representing enforcement of both orderings. This can occur if, for example, there are two fences or MORs back-to-back in a program. We define this operation as the *join* operator ($\vee$). A join operation is intuitively similar to a max operation, except that the result may not be equal to any one of the inputs, because comparison is not totally

Power `lwsync`

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO St	BC St
PO Ld	✓	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓	✓
PO St	—	—	—	$\checkmark_N$	$\checkmark_N$
AC St	—	—	—	$\checkmark_N$	$\checkmark_N$

<

Power `sync`

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO St	BC St
PO Ld	✓	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓	✓
PO St	✓	✓	✓	$\checkmark_S$	$\checkmark_S$
AC St	✓	✓	✓	$\checkmark_S$	$\checkmark_S$

Figure 3.10: MOST comparison example: Power `lwsync` is strictly weaker than Power `sync`, since each entry in the MOST for `lwsync` is weaker than or equal to its counterpart in the `sync` MOST, and at least one comparison is strict.

ordered. Instead, the join produces a new MOST which is at least as strong (in terms of $\geq$ above) as each of the input MOSTs. The calculation of a join is also defined cell-by-cell; each cell in the result MOST must be an ordering strength which implies the strength levels in the corresponding cells of both input tables. In other words, if $A \vee B = C$, then C must satisfy $C \geq A$ and $C \geq B$.

Lastly, *subtraction* ($-$) produces a MOST which specifies the orderings which are enforced by the first MOST but not by the second. Conceptually, this corresponds to a scenario in which a certain set of orderings is required, but a particular MOR may only enforce some subset of those orderings; subtraction of these two MOSTs produces the set of required orderings that remain unenforced. Again, this can be calculated in a cell-by-cell manner.

Finally, it would be easy to consider other ArMOR comparison operators which could be defined analogously. For example, there may be some scenarios in which it becomes useful to define a meet ($\wedge$) operator, which would perform the opposite operation of the join operator: it would produce a MOST which is at least as weak as each input. Because such operators were not needed in any analyses in this thesis, we do not consider them further.

Power PPO

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO+ SA St	PO+ DA St	BC St
PO Ld	✓	—	—	✓	—	—
AC Ld	—	—	—	—	—	—
PO St	$\checkmark_L$	—	—	$\checkmark_L$	—	—
AC St	—	—	—	—	—	—

$\vee$

Power `lwsync`

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO+ SA St	PO+ DA St	BC St
PO Ld	✓	✓	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓	✓	✓
PO St	—	—	—	$\checkmark_N$	$\checkmark_N$	$\checkmark_N$
AC St	—	—	—	$\checkmark_N$	$\checkmark_N$	$\checkmark_N$

$=$

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO+ SA St	PO+ DA St	BC St
PO Ld	✓	✓	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓	✓	✓
PO St	$\checkmark_L$	—	—	$\checkmark_N$	$\checkmark_N$	$\checkmark_N$
AC St	—	—	—	$\checkmark_N$	$\checkmark_N$	$\checkmark_N$

Figure 3.11: Power PPO $\vee$ `lwsync`, where "$\vee$" is the MOST join operator, produces the complete set of orderings enforced when a `lwsync` instruction is executed.

3.4.3 ArMOR Comparison Examples

As a relatively simple example, consider a comparison of the two MOSTs of Figure 3.8. By comparing each pair of corresponding cells in the table, it is clear that `lwsync` $<$ `sync`: every cell in the `sync` MOST is at least as strong as the corresponding cell in the `lwsync` MOST, and some comparisons are strict. In this case, the join ($\vee$) of the two tables is equivalent to the `sync` MOST.

A less trivial example of a join operation is shown in Figure 3.11. This figure shows Power PPO being joined with the Power `lwsync` fence. Notably, the latter is not strictly stronger than the former; in particular, the ordering of a store followed by a load to the same address is maintained (with strength $\checkmark_L$) by PPO but not at all by `lwsync`. Therefore, to calcualte the complete set of orderings maintained when an

TSO PPO

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO+ SA St	PO+ DA St	BC St
PO Ld	✓	✓	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓	✓	✓
PO St	$\checkmark_L$	—	✓	$\checkmark_M$	$\checkmark_M$	$\checkmark_M$
AC St	✓	✓	✓	$\checkmark_M$	$\checkmark_M$	$\checkmark_M$

Power PPO

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO+ SA St	PO+ DA St	BC St
PO Ld	✓	—	—	✓	—	—
AC Ld	—	—	—	—	—	—
PO St	$\checkmark_L$	—	—	$\checkmark_L$	—	—
AC St	—	—	—	—	—	—

−

Power `lwsync`

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO+ SA St	PO+ DA St	BC St
PO Ld	✓	✓	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓	✓	✓
PO St	—	—	—	$\checkmark_N$	$\checkmark_N$	$\checkmark_N$
AC St	—	—	—	$\checkmark_N$	$\checkmark_N$	$\checkmark_N$

−

=

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO+ SA St	PO+ DA St	BC St
PO Ld	—	—	—	—	—	—
AC Ld	—	—	—	—	—	—
PO St	—	—	✓	$\checkmark_M - \checkmark_N$	$\checkmark_M - \checkmark_N$	$\checkmark_M - \checkmark_N$
AC St	✓	✓	✓	$\checkmark_M - \checkmark_N$	$\checkmark_M - \checkmark_N$	$\checkmark_M - \checkmark_N$

Figure 3.12: TSO PPO, properly refined (Figure 3.9b) − Power `lwsync` (Figure 3.8), where "−" is the MOST subtraction operator. The shaded cell highlights the ordering that distinguishes the two cases in Figure 3.2.

`lwsync` is executed, the two MOSTs can be joined together, resulting in the MOST shown in the figure.

Finally, as an example of MOST subtraction, we return to the TSO PPO vs. Power `lwsync` example shown earlier in the chapter. To show explicitly how the two differ, Power `lwsync` can be subtracted from TSO PPO (once the latter is properly refined). This result is shown in Figure 3.12. The fact that the subtraction result is

non-empty not only shows that `lwsync` enforces fewer orderings than TSO requires; it also shows exactly *which* orderings are unenforced.

A major benefit of ArMOR is that it supports manipulations that can be performed entirely algorithmically. In the next chapter, we will use ArMOR to automatically derive the designs of consistency model translation modules called shims, given only the set of MOSTs used by the input and output memory consistency models.

3.5 Related Work

Model-Independent Specification Frameworks. Adve and Gharachorloo provide a comprehensive survey [AG96] of early attempts to define various consistency models. Gharachorloo studied many of the architectures analyzed in this thesis, and he developed a framework which covered all of those models [Gha95]. However, some of the models in that thesis are no longer in use, and others have undergone dramatic changes since that time. In particular, cumulativity was not yet a feature of the Power architecture (which was at the time was called PowerPC). Furthermore, this work took place prior to more recent formalization efforts that have shone light on other unexpected corner cases not yet considered at the time.

Other authors have proposed frameworks which unify some consistency models, but it is not clear whether these models apply to all modern relaxed hardware models [SN04, YGLS04]. Mador-Haim et al. analyze and classify a family of 90 single- and multi-copy atomic memory models; they distinguish same- vs. different-address relationships, but locally-enforced orderings, cumulativity, and non-multi-copy-atomicity are not handled by their framework [MHAM11]. Recently, Alglave et al. developed the general-purpose `herd` software suite and the `cats` language for specifying single-event axiomatic memory models [AMT14, AMS+12]. However, this framework does not allow for easy comparison of different models in the way that ArMOR does.

Petri et al. use an approach very similar to MOSTs to describe the ordering requirements of an abstracted version of Java bytecode which they call "cookbook high" [PVJ15]. They analyze an abstracted subset of Java containing volatile and normal loads and stores, and they describe the ordering requirements using a large table which vaguely resembles a MOST. Their encoding, however, can only be interpreted in the context of their specific model. In particular, their specification defines orderings that need to be enforced within the "temporary store", a feature of their particular operational model which is used to model partial and incremental visibility of memory accesses, and it refers explicitly to orderings on speculative accesses and futures which exist only within the formalism.

Defining Fence/MOR Behavior. Both operational and axiomatic models often "hard-code" fence behavior into the model in some way, in the sense that the semantics of each type of fence or ordering mechanism are built into the models in ways that cannot easily be changed [Alg12, MHMS$^+$12, OSS09a, SSA$^+$11]. Owens et al. explicitly state that the behavior of `mfence` on an x86-TSO processor is to flush an idealized store buffer. The Power operational model of Sarkar et al. omits store buffers in favor of more abstract sets of visibility events, and fences have both visibility events and (in the case of the `sync` fence only) acknowledgment events [SSA$^+$11]. Adding new fences would likewise require new fence-specific operational events to be added. The axiomatic models of Mador-Haim et al. and of Alglave et al. are somewhat more flexible in the sense that axioms can simply be added, modified, or removed [Alg12, MHMS$^+$12]. However, even with this flexibility, the semantics of less common fences such as `eieio` and `dsb` are left vague, as the industry specifications are unclear on many of the subtle details of their behaviors. To the best of our knowledge, no existing model specifies fence types and ordering specifications in a way that is sufficiently general and architecture-independent that inter-architecture comparison can be rigorously performed.

Applicability to Related Topics. Recent work has also explored the application of consistency models to non-volatile storage [PCW14]. We see ArMOR as applicable to memory persistency model analysis as well.

3.6 Chapter Summary

As Chapter 4 will show, MOST notation is being useful across a broad range of compilation and translation tasks including static compilation, JIT compilation, dynamic binary translation, and more. ArMOR highlights the pros and cons of different choices of fences and MORs, and we use ArMOR to provide insights that can assist in exploring memory system design tradeoffs in future heterogeneous systems.

Appendix A presents a gallery of MOST-based definitions for well-known hardware memory consistency models.

Chapter 4

Dynamic Translation Between Memory Consistency Models[1]

While the previous chapter demonstrated the MOST notation and the ArMOR comparison and manipulation framework, this chapter presents an in-depth case study of the new kinds of technologies that ArMOR can enable.

4.1 Introduction

As earlier chapters have discussed, computer architecture is undergoing a dramatic shift away from homogeneous multicores and towards increasing microarchitectural and architectural heterogeneity [CRDI07, Gre11, PCC+14, Shi, Top15]. In the context of memory consistency models, heterogeneity brings with it a number of challenges: how to compile from a given software model onto a given hardware model, how to design memory model-aware intermediate representations (e.g., LLVM IR, NVIDIA PTX), how to dynamically migrate code from one ISA to another, and so on. All of these use cases require the direct comparison of memory ordering mechanisms between

[1]Some of the work in this chapter was performed in collaboration with fellow graduate student Caroline Trippel and other contributors [LTPM15a].

different architectures and/or between different microarchitectures, and so all of the above could serve as interesting use cases for the ArMOR framework developed in the previous chapter. In this chapter, we highlight one use case in particular.

Recent work has demonstrated the performance and/or power benefits of performing dynamic binary translation across ISAs and/or microarchitectures [DVT12, VT14]. However, this previous work focused on opcode-for-opcode translation and memory layout issues; it did not address memory consistency models. Inter-consistency model translation has previously been studied only for specific cases such as SC→TSO [DMT13, VN11].

In this chapter, we fill this gap by showing how ArMOR can be used to automatically derive the designs of self-contained translation modules called *shims*. Shims dynamically adapt code compiled for one memory model to execute on hardware implementing another, *without recompilation or offline code analysis*, by dynamically injecting fences or other enforcement mechanisms as needed into a code stream. Other use cases for shims could include removing redundant fences to optimize performance or using programmable shims as a means of allowing memory-accessing IP blocks to be built independently of the reordering properties of the underlying infrastructure (e.g., the network-on-chip). In the future, we envision shims being used in JIT compilers [Khr, NVI13a], dynamic binary translators [DVT12, VT14], dynamic code optimizers [NVIb], and updates to reprogrammable microcode [Int13a], with the latter used to fix implementation bugs [ABD+15].

Although translation results in some overhead, we demonstrate that this cost can be outweighed by the benefits of migrating to faster or more power-efficient hardware. When memory models are sufficiently compatible, we demonstrate that the performance overhead of implementing shims in hardware can be as low as 10-77%. In other cases, our experiments motivate ways in which instruction sets could be augmented in ways that could mitigate much of the overheads. For example, we

propose the addition of finer-grained fence types and/or the maintenance of otherwise-redundant consistency metadata within the instruction set definition.

The rest of this chapter is organized as follows. Section 4.2 presents a motivating example. Section 4.3 describes the operation of translation shims. Section 4.4 presents an experimental evaluation methodology, and Section 4.5 presents the results of these experiments. Section 4.6 then presents some takeaways from the experiments. Section 4.7 presents some related work, and Section 4.8 concludes. Finally, a gallery of shim designs is presented in Appendix B of this thesis, and the code used to generate this appendix is open source [Lus15].

4.2 Motivating Example

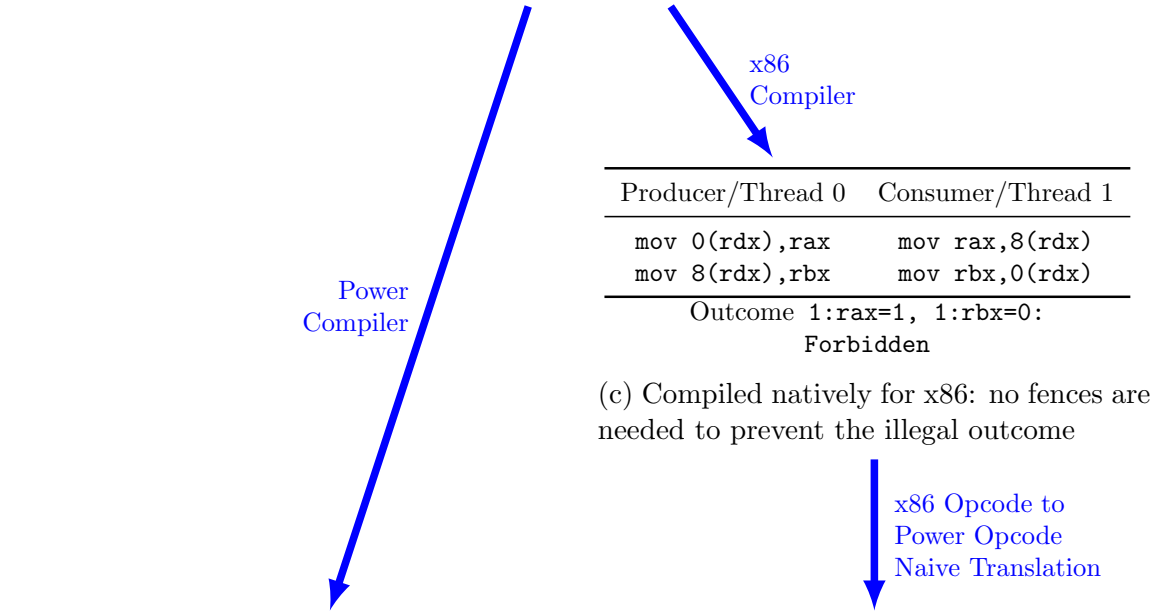
Figure 4.1a shows C11 source code for the `mp` (message passing) litmus test. For this test, the C11 memory ordering rules specify that if the consumer reads 1 from y, then it must also return 1 from x. In a traditional scenario, the compiler ensures that all of the C11 ordering rules within each thread are respected by the generated assembly code with respect to the relevant hardware memory model. This generally occurs by looking up the architecture-specific implementations of the software synchronization constructs in a pre-calculated table like the one in Figure 2.1. On Power, the orderings are enforced by inserting `lwsync` fences, as shown in Figure 4.1b. On x86-TSO, as Figure 4.1c shows, no fences are needed.

Problems arise if one tries to perform naive binary translation of the x86 code to execute on the Power architecture. Naive opcode-for-opcode translation would produce the code in Figure 4.1d. Unfortunately, because the source x86 code lacks fences, the translated code also lacks fences, meaning that the extra `lwsync` fences required to prevent the bad outcome of the `mp` litmus test on Power are missing. This demonstrates that *if cross-ISA binary translation techniques do not account for the*

```
// Producer/Thread 0
*x = 1;
atomic_store_explicit(&y, 1, memory_order_release);

// Consumer/Thread 1
if (atomic_load_explicit(&y, 1, memory_order_acquire))
  assert(*x != 0);
```

(a) C11 source code for `mp`



Producer/Thread 0	Consumer/Thread 1
`mov 0(rdx),rax`	`mov rax,8(rdx)`
`mov 8(rdx),rbx`	`mov rbx,0(rdx)`

Outcome `1:rax=1, 1:rbx=0:` `Forbidden`

(c) Compiled natively for x86: no fences are needed to prevent the illegal outcome

Producer/Thread 0	Consumer/Thread 1
`stw r1,0(r2)`	`lwz r1,8(r2)`
`lwsync`	`lwsync`
`stw r3,8(r2)`	`lwz r3,0(r2)`

Outcome `1:r1=1, 1:r3=0:` `Forbidden`

(b) Compiled natively for Power: fences prevent the illegal outcome

Producer/Thread 0	Consumer/Thread 1
`stw r1,0(r2)`	`lwz r1,8(r2)`
`stw r3,8(r2)`	`lwz r3,0(r2)`

Outcome `1:r1=1, 1:r3=0`

× *Observable* ×

(d) Compiled for x86 and translated to Power. Since x86 code does not contain fences, it becomes the job of the DBT engine to insert fences. Otherwise, the bad outcome becomes observable.

Figure 4.1: A compiler targeting either architecture directly would produce correct code. However, binary translation that does not account for differences in consistency models would lead to the invalid outcome becoming observable.

consistency model, the resulting code could produce illegal outcomes. The goal of this chapter is therefore to generate translator shims which automatically and dynamically determine where to insert MORs and which MORs to insert, without requiring offline code analysis.

The above example motivates the need for cross-layer analysis, mapping, and compilation frameworks to aware of the memory models of each layer. It also motivates the need for a memory model analysis framework which is flexible enough to reason about both the source and the target model concurrently. In general, there are many scenarios to which this applies: compilation from one model to another (as in Section 2.4), static and/or dynamic binary translation between models [DVT12, LCM+05, Qem15, VT14], dynamic optimization [NVIb, DGB+03], and so on. This chapter performs an in-depth case study of one of these scenarios: dynamic binary translation from one hardware memory model to another.

4.3 Cross-MCM Dynamic Binary Translation

In this section, we first explain the goal and design criteria for dynamic binary translator shims. We then explain how ArMOR and MOSTs can be used to automatically derive the basic designs of shims. Finally, we discuss a number of design questions that come up during the implementation process.

4.3.1 High-Level Operation

Conceptually, we consider dynamic binary translation to take place on a *stream*: an ordered sequence of memory operations (loads, stores, or fences) passing through some particular point in a processor or IP core. The specific format of the stream operations depends on the location where the translation is conducted. Streams may carry macroops, microops, or whatever other form operations may take at the chosen

location. A stream may also carry implicit (via preserved program order) or explicit (via fences or other MORs) ordering requirements on its memory operations. We refer to incoming (newer) operations as *upstream operations* and outgoing (older) operations as *downstream operations*.

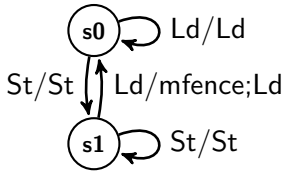
The goal of a *shim* is to map each incoming upstream operation into zero or more downstream operations which are strong enough to enforce the memory ordering requirements of the upstream operation. More specifically, to translate an explicit upstream MOR such as a fence, the shim must emit zero or more downstream operations which combine to implement all of the ordering requirements specified by that fence. Likewise, to handle implicit upstream ordering requirements, the shim must enforce any upstream PPO requirements that are not enforced by downstream PPO.

An overly-conservative (and hence low-performing) but correct baseline would be to insert the strongest possible fence between each pair of instructions. In most cases, this is sufficient to restore sequential consistency[2], let alone the requirements of the source architecture. However, this approach is overkill, as many inserted fences would be redundant and unnecessary. Instead, we build shims as finite state machines (FSMs) which insert MORs lazily—just before they are actually needed. The design and derivation of these FSMs is the subject of the next subsection.

4.3.2 Shim Finite State Machines: Overview

Conceptually, shims are finite state machines in which downstream MOR insertion takes place while traversing certain state transitions. Each FSM state represents a particular set of *pending ordering requirements*. An ordering requirement $a \rightarrow b$ of some strength (as defined in Section 3.3.1) is said to be pending if 1) the ordering is not enforced by the preserved program order (i.e., default ordering enforcement) of the target architecture, 2) an access of type a has been observed, and 3) no other

[2]Such restorations are not universally achievable; for example, Itanium unordered accesses cannot be made sequentially consistent [Int10].



(a) FSM (in agreement with previous work [DMT13, VN11])

Notation	Meaning
x/y	On an incoming upstream operation x, send y downstream
x/y;z	On an incoming upstream operation x, send y followed by z downstream

(b) Key

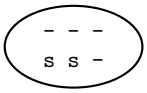
Figure 4.2: Shim FSM for SC upstream and TSO downstream. Although this particular case has been studied before (and hence serves as a sanity check), analogous shims can be generated for any pairing of source and target memory models that are defined using MOSTs.

ordering mechanism (fence or otherwise) that would enforce the $a \rightarrow b$ ordering has been inserted between a and b. Together, these conditions imply that before an access of type b is emitted downstream, some explicit ordering mechanism must be inserted downstream to enforce the $a \rightarrow b$ ordering in a sufficiently strong manner. As mentioned above, shims insert MORs lazily in order to avoid the insertion of redundant MORs. This laziness is implemented by emitting MORs downstream along the transitions between certain pairs of FSM states, as described below.

As an example, consider a scenario in which a piece of code was compiled to run on a sequentially consistent processor, but it will instead be executed on a processor only guaranteeing to enforce TSO by default. This particular case has been studied before in a more restricted context [DMT13, VN11]. In this way, it serves as a useful sanity check for our algorithm. The most prominent difference between SC and TSO is that TSO allows store→load reordering while SC does not. Previous work therefore concluded that to restore SC on a TSO processor, `mfence` operations must be inserted such that there is (at least) one `mfence` between each store and any subsequent load.

	PO+SA Loads	PO+DA Loads	PO Stores
PO Loads	—	—	—
PO Stores	s	s	—

(a) Example pending orderings table. Pending ordering tables match the layout and design of MOSTs, but they track orderings which are *pending* rather than orderings which are required.



(b) Corresponding depiction within the shim FSM figures used in this thesis

Label	Description
s	An ordering of strength $\checkmark_S$ is pending
m	An ordering of strength $\checkmark_M$ is pending
n	An ordering of strength $\checkmark_N$ is pending
l	An ordering of strength $\checkmark_L$ is pending
✓	An ordering of strength ✓ is pending
—	No ordering is pending

(c) Pending orderings legend (derived from Fig. 3.4)

Figure 4.3: Pending ordering tables, which are used to described states within shim FSMs

Figure 4.2 shows the FSM translating SC to TSO. The shim for this scenario has two states: a starting state s0 and a second state s1. After any load, the FSM transitions to state s0. After any store, the FSM transitions to state s1. For three of the four transitions, the shim does not need to insert any new MORs downstream; it simply passes the upstream operation through unmodified. For the last transition, when the FSM is in state s1 and a load arrives upstream, the shim inserts an `mfence` operation downstream before passing the store through and then transitioning to s0. The fact that a `mfence` is inserted for this particular state and input fortunately matches the intuition of previous work.

Although the particular case of SC-to-TSO translation has been studied by previous work, our use of MOSTs and the ArMOR framework provides two key advantages over previous work. First, they are flexible enough to apply to other models beyond SC and TSO. Second, they can be automatically generated for any source/target pairing. All that is needed is specification of the source and target memory models using MOSTs, as discussed in Chapter 3. We discuss the generation algorithm below.

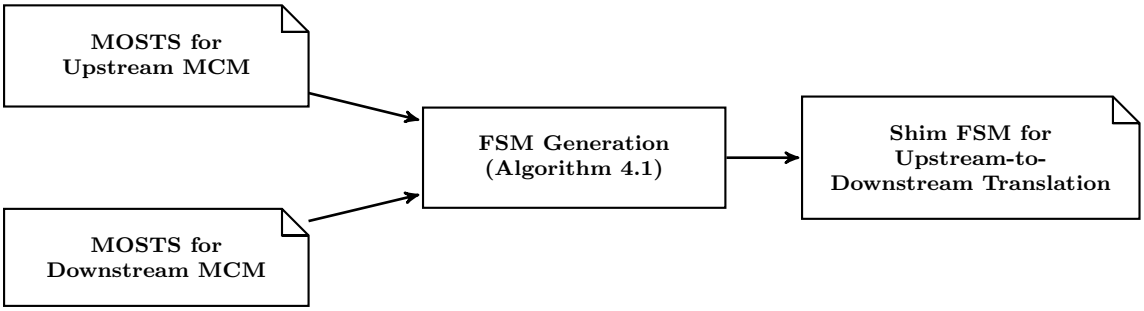


Figure 4.4: Shim FSM generation process overview

Later in this chapter, the analysis of Section 4.5 analyzes a broader set of cases in detail, and then Appendix B presents a large gallery of shim designs for a wide variety of upstream and downstream memory models.

4.3.3 Shim FSM Generation

This subsection describes the process by which shims FSMs are generated. The following subsection presents a detailed example of this process in action. At a high level, the procedure is as shown in Figure 4.4: the upstream and downstream MCMs are defined using sets of MOSTs, and the MOSTs are fed into the FSM generation algorithm described below. The output of this process is the shim FSM translating between the two models.

Each shim FSM state represents a particular *pending ordering table*. A pending ordering table tracks whether the each type of upstream operations *has actually been observed* since any relevant earlier fence(s) or ordering mechanism(s). Pending ordering tables are in a sense the inverses of MOSTs; rather than specifying which orderings are required, they specify the orderings that have not (yet) been enforced. Nevertheless, as the layout and contents are analogous, we use similar notations for both. As shown in Figure 4.3, we depict pending orderings of a given strength with the lowercase equivalent of the uppercase notation from Figure 3.4. Each cell in the

table tracks whether an ordering of the access type in the corresponding row heading is pending with respect to the access type in the corresponding column heading.

Shim FSMs are generated by calculating all states (i.e., all pending ordering tables) reachable from the given start state. The start state represents the empty pending ordering table—the table in which no orderings are marked as pending. Upon receiving an input (i.e., an upstream operation), the shim first checks whether any new MORs must be inserted downstream before passing the input through. It then calculates and transitions to the next state, reflecting the fact that the orderings considered pending may have changed due to the receipt of a new type of input and/or the insertion of a new ordering enforcement MOR into the stream.

The detailed shim FSM state transition function is given in Algorithm 4.1. We first describe the general procedure and give an example demonstrating the process. After that, we discuss broader design questions and address details that come up when implementing shims in practice.

State Transition Function. To determine which pending orderings (if any) need to be enforced, the algorithm does two things. First, it searches the pending ordering table for that state to find a column corresponding to the input operation type. This represents the fact that the input operation would effectively serve as the b component for any table cell tracking a pending $a \rightarrow b$ ordering. All cells marked pending in the column(s) in question must be enforced. The input operation may not have an associated column (e.g., fences are not often listed as columns, because fence→fence orderings need not always be explicitly tracked), in which case this step returns the empty MOST. Second, the algorithm also checks whether the input operation itself has an associated MOST (e.g., again, as fences would), and if so, it adds those to the set of operations to be enforced. This addition takes place using the join ($\vee$) operator of Section 3.4.2. The result of this addition is a MOST

Algorithm 4.1 Shim FSM Transition Function

Function: NextState(currentState, op):

```
// Enforce pending orderings in relevant column(s)
orderingsToEnforce = MOST(op) ∨ KeepColumn(currentState, op) ∨ assumedReqs

// Mark relevant orderings in relevant row(s) pending
newOrderings = KeepRow(upstreamPPO − downstreamPPO, op)

// Find a MOR which enforces the orderingsToEnforce, if necessary
insertedMOR = WeakestSufficientMOR(orderingsToEnforce)

// propagated = pending and not enforced by the inserted MOR
propagatedOrderings = state − MOST(insertedMOR)

// Join old and new orderings to calculate the next state
nextState = propagatedOrderings ∨ newOrderings

// If necessary, insert the new MOR before passing the input operation through
if insertedMOR ≠ ∅ then
    Emit(insertedMOR)

// Pass the input operation through
Emit(op)

// Done; transition to the next state
return nextState

// Helper functions used above:
```

$$\text{KeepColumn}(s, col)[i][j] = \begin{cases} - & j \neq col \\ s[i][j] & j = col \end{cases}$$

$$\text{KeepRow}(s, row)[i][j] = \begin{cases} - & i \neq row \\ s[i][j] & i = row \end{cases}$$

`orderingsToEnforce` describing the pending orderings that must be enforced. We omit a discussion of `assumedReqs` for the moment; see Section 4.3.5 for details.

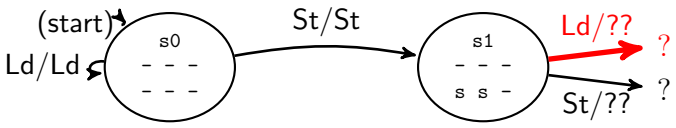
If any pending orderings do need to be enforced, the algorithm searches for the weakest (i.e., fastest) MOR which is sufficiently strong (according to the $\geq$ operator from Section 3.4.2), and it inserts that MOR downstream. This has the effect of subtracting (as defined in Section 3.4.2) the enforced orderings from the current set of pending orderings. This subtraction produces a new intermediate pending orderings table called `propagatedOrderings`.

Next, the shim FSM algorithm also then tracks the fact that each input operation will serve as the a component for future $a \rightarrow b$ orderings. This is tracked by marking cells in the row corresponding to the input operation type as pending if, for that particular cell, the upstream PPO is stronger than the downstream PPO (according to the $\geq$ operator of Section 3.4.2). This produces a second intermediate pending ordering table `newOrderings`. The two intermediate tables are then joined to form the pending ordering table `nextState` representing the next state of the FSM.

Lastly, the operation(s) can be emitted downstream. First, if the above steps that a new MOR needed to be inserted downstream to enforce some pending ordering(s), this inserted MOR is emitted downstream first. Second, the original upstream input operation is propagated downstream. Once, this is completed, the transition is complete, the state has been updated, and the next iteration can begin.

In certain degenerate cases, the above algorithm may produce a shim that consists of a FSM with only a single state. This corresponds to cases in which the shim behaves the same in all scenarios, and hence the FSM can be condensed into a single state to save performance/power/area overhead. As we show in later sections, this degenerate case does occur in practice. We therefore refer to shims with more than one distinct state as *stateful shims* and to those which reduce to a single state as *stateless shims*.

Note that while the description above presents the conceptual overview of shim FSM generation, all of the necessary calculation and analysis described below can be performed *offline and in advance*. Furthermore, the state space of the shim FSMs is small enough that the space can be explored completely in just seconds. As Section 4.5.1 shows, shim FSMs generally end up with a very small number of states, and hence they can be both derived and implemented very cheaply.



(a) Assume at the start of this example that state s0 has already been explored, and that state s1 is currently being explored.

	PO+SA Loads	PO+DA Loads	PO St
PO Ld	—	—	—
PO St	s	s	—

(b) The pending orderings table state s1 in Figure 4.2. During the shim generation process, state s1 represents the above pending orderings table.

	PO+SA Loads	PO+DA Loads	PO St
PO Ld	—	—	—
PO St	—	s	—

(c) Normal loads do not have their own MOSTs, so `orderingsToEnforce` is formed by keeping the column of Figure 4.5b which corresponds to the input operation.

	PO+SA Loads	PO+DA Loads	PO St
PO Ld	✓	✓	✓
PO St	s	s	s

(d) The MOST for `mfence` is strong enough to enforce the pending orderings of Figure 4.5c. Since `mfence` is the only choice, it is also the `WeakestSufficientMOR` and hence the `insertedMOR`.

	PO+SA Loads	PO+DA Loads	PO St
PO Ld	—	—	—
PO St	—	—	—

(e) `newOrderings` is calculated by keeping the "PO Ld" row of the difference between SC PPO and TSO PPO. In this case, the input rows are identical (Figure 3.7, after proper refinement), and so the difference is empty.

	PO+SA Loads	PO+DA Loads	PO St
PO Ld	—	—	—
PO St	—	—	—

(f) `propagatedOrderings` is calculated by subtracting the MOST for the `insertedMOR` from the current state. In this case, the subtraction produces an empty MOST.

	PO+SA Loads	PO+DA Loads	PO St
PO Ld	—	—	—
PO St	—	—	—

(g) `nextState` is calculated by joining Figures 4.5e and 4.5f. In this particular case, the `nextState` matches the already-explored state s0.

(h) The above analysis adds one new transition to the FSM. From here, the algorithm continues exploring all unexplored possibilities until the FSM converges.

Figure 4.5: Example of shim FSM generation (Algorithm 4.1) in action for the case of translating SC to TSO (as in Figure 4.2).

4.3.4 Shim FSM Generation Example

As an example of how Algorithm 4.1 is used to generate shim FSMs, consider Figure 4.5. This figure continues the example started in Figure 4.2. It shows how MOSTs can be used to derive one particular transition in that FSM. Other transitions in the FSM are derived analogously.

This example starts from the partially-explored FSM of Figure 4.5a, and it calculates what should happen when the FSM is in state s1 and a load to a not-yet-seen address is received. The pending orderings table for state s1 is shown in Figure 4.5b. The algorithm searches the column of this table to find orderings that need to be enforced before the load is propagated (Figure 4.5c). Since there is at least one such pending ordering, the algorithm searches for a sufficiently-strong MOR to insert. In this case, there is one option: `mfence` (Figure 4.5d). An `mfence` is therefore emitted downstream before the load is passed through the shim.

To calculate the next state, the algorithm performs a similar set of calculations. First, it determines whether any new orderings need to be marked as pending. Since the "PO Ld" row of SC PPO and TSO PPO are identical (once they are properly refined), their difference is the empty MOST, and so no orderings need to be enforced (Figure 4.5e). Second, it determines whether any old pending orderings need to be propagated. Intuitively, the emitted `mfence` has the effect of enforcing all orderings, and so none need to be propagated. Using MOSTs, the subtraction of the `mfence` MOST from the current state indicates the same thing (Figure 4.5f). Third, the results of the two previous steps are joined together to form the next state (Figure 4.5g).

In this particular example, the next state happens to correspond to a state that has already been explored: state s0. Figure 4.5h therefore adds a transition from s1 to s0, indicating that upon receiving a load, an `mfence` should be emitted downstream before the load is passed through, and that the next state is s1. In other cases, the edge may point to an as-of-yet unexplored state, in which case the new state would

be created and the state space exploration algorithm would recursively explore all possible transitions from that new state as well. Finishing this example, running Algorithm 4.1 to explore the final unexplored edge in Figure 4.5h would create the self-loop from/to s1 as seen in Figure 4.2, and this would complete the shim generation process.

4.3.5 Design Considerations

Handling Non-Visible Operations. One subtlety in Algorithm 4.1 is the use of assumed pending ordering requirements, or "assumedReqs". This property handles the practical need to support situations in which accesses from certain rows and/or columns are not directly observable by the shim. Most commonly, this refers to situations in which a shim within one core cannot see some or all of the events happening at other cores, even though cumulative fences (Section 3.3.3), for example, require cross-core orderings to be tracked and enforced. Other similar situations are discussed below. Regardless of the reason, events which cannot be directly observed must be conservatively assumed to be pending ordering enforcement.

The "assumedReqs" pending orderings table contains a set of orderings which are permanently marked as pending and which are added during every transition. This approach allows shims to translate correctly even when the necessary information is only partially available. If the information were to somehow become available later (e.g., in a subsequent microarchitectural revision), the "assumedReqs" could be weakened accordingly, and a new (and likely more efficient) shim could be regenerated.

Special Case Optimization. We include one special-case optimization in our evaluations: we allow a store→store ordering of strength $\checkmark_S$ to be enforced by a MOR with enforcing store→store ordering of strength $\checkmark_M$ *if* store→load ordering of strength ✓ is marked as pending. Intuitively, this captures the notion that the only difference between single- and multi-copy atomicity is that forwarding from a

local store buffer is permitted only in the latter. This optimization, which is used implicitly in previous work [DMT13, VN11], makes shims slightly lazier and hence improves performance.

Microarchitectural Placement. ArMOR requires that a stream be sorted into a legal visibility order so that preserved program order needs can be detected. This means that when shims are implemented in hardware, their placement within the pipeline matters. Placing a shim too early in the pipeline may render it unable to track same/different address dependencies (Section 3.3.2), as the addresses may not have been resolved by that point. Placing it at a later location through which non-memory instructions do not pass may prevent it from being able to observe information such as inter-instruction dependencies. In such cases, these unobservable dependencies would have to conservatively be included into "assumedReqs" (Algorithm 4.1).

Likewise, shims may not be able to observe any accesses made by other cores, as the mechanisms that would be needed to enable such visibility would likely have large performance/power/area overheads in software and/or in hardware. To handle such cases, any orderings which explicitly refer to accesses being made by other cores (e.g., for cumulative fences) would need to be conservatively included into "assumedReqs" as well.

Our hardware evaluation (Section 4.4.2) places a shim into the issue stage of a pipeline, as all necessary information about accesses in program order is observable at that point. Our software dynamic binary translation-based evaluation operates on the full stream of instructions in their original program order.

Microarchitecture-Level Integration. The analysis in this section so far has been at the level of the architecture—i.e., at the level of the hardware specification. However, ArMOR and the shim FSM generation algorithm easily adapt to situations in which the set of fences provided microarchitecturally varies from the architectural specification. Section 4.4.2 provides one example: the pipeline analyzed in that sec-

tion implements a finer-grained set of fence choices than would otherwise be available according to the architecture specification. We therefore adapt the MOSTs and the analysis of that section to include those fences.

Laziness. Lazy insertion is not the only possible design approach. More eager insertion could make it easier to hide the latency of inserted fences, but it may also result in inserting a larger number of fences. Our experience is that the benefits of laziness outweigh the small potential latency hiding of eagerness.

Atomic Instructions. Atomic (i.e., uninterruptible) instructions such as compare-and-swap can be easily added into our model. If atomics are considered a separate class of instruction from loads and stores, they would simply form new MOST rows and columns. Alternatively, if atomics are treated as a bundled load-store pair, Algorithm 4.1 could be modified to look up multiple rows and columns for such instructions, rather than just one. Either solution is viable as long as the ordering implications of such operations are correctly specified.

Downstream Non-Fence MORs. Downstream MORs need not be fences. It is possible, for example, to use lightweight MORs such as ARM/Power address dependencies as well. Doing so would require more than just fence insertion; it would also require rewriting instruction operands, but dynamic binary translators already do this regularly [LCM+05].

Stream Interruptions. Streams may be interrupted by events such as context switches and hence lose their state. A conservative solution is to simply insert a full fence instruction and then return to the start state. A more aggressive solution would be to jump to a state which marks all orderings not enforced by the downstream PPO as pending. In either case, ArMOR's fundamental operation remains the same.

Uniqueness. There may be multiple valid shim FSMs for a given scenario, depending on the search order of WeakestSufficientMOR in Algorithm 4.1, whether the emission of multiple downstream fences are allowed, and so on. We highlight one

case study in our results section: although the MOR specification of the x86 `mfence` instruction and LOCK prefix are identical, their performance costs are not. In that case, the performance analysis becomes the deciding factor for which MOR to use.

Dynamically Changing Partition Subsets. Algorithm 4.1 as shown assumes that the categorization of a given instruction into a particular row and column is based only on static properties of that instruction. In particular, if an access were to change from being in one partition to being in another, Algorithm 4.1 as shown would not be able to track that movement. However, the assumption that the classification stays constant is not always true. For example, AMD GPUs contain a fence `s_waitcnt <n>` which only enforces ordering with respect to accesses other than the most recent n. If the classification of a memory access can change, the state transition function must be modified to account for such changes. In practice, few MORs are defined this way, and so a cheaper option may be to overapproximate the state and avoid the need for such dynamic reclassification altogether.

Insufficient Target MORs. In almost all cases, we assume there is a sufficiently strong set of downstream MORs to support any upstream ordering requirement. In rare cases, this may not actually hold, and then Algorithm 4.1 would fail when searching for the WeakestSufficientMOR. This is not a limitation of ArMOR, but rather an inherent limitation of certain architectures themselves. For example, sequential consistency cannot be enforced onto Itanium relaxed loads and stores, even with `mf` fences interleaved [Int10], and NVIDIA GPUs simply do not provide any mechanism for enforcing cumulativity or atomicity [NVI13b]. In practice, there are often workarounds such as replacing Itanium relaxed loads and stores with `ld.acq` and `st.rel`, respectively. For all architectures surveyed in this thesis other than GPUs, a sufficiently strong MOR is in fact present, and so this limitation does not appear.

Property	Real System	Simulator
System	8-core	4-core
CPU	Xeon X7560	gem5 O3
Frequency	2.27 GHz	2.0 GHz
Pipeline	OoO	OoO
L1I Cache	32kB, private	32kB, private
L1D Cache	16kB, private	64kB, private
L2 Cache	256kB, private	2MB, shared
Cache coherence protocol	MESI	MOESI_hammer
Memory timing model	N/A	Ruby

Table 4.1: System configurations

Speculation. ArMOR does not inhibit the use of speculative ordering enforcement techniques [DMT13, GGH91], as long as these techniques maintain the architecturally-required behaviors.

4.4 Evaluation Methodology

In this section, we describe our evaluation of the translation shims. We first provide a characterization of the breadth of applicability of ArMOR by generating shims for a number of upstream and downstream models. We then evaluate the performance of a subset of these models. We break our performance evaluation into two parts. We first implement ArMOR shims as software Pintools [LCM+05]. With near-native speeds, this approach allows for rapid exploration of various design possibilities. We then evaluate ArMOR in hardware by inserting shims into the gem5 O3 simulator pipeline [BBB+11].

4.4.1 Pintool-Based Exploration Methodology

Software-based dynamic binary translation can be used by architects to explore the performance impact of different hardware ordering requirements, fence implementations, or translation approaches prior to their being hardened into a processor. It can also be used as an implementation in and of itself, whether it be standalone, within

an emulator, within a virtual machine manager, or as a component within any other tool. We use the Pintool approach to quantify the performance impact of statefulness (as opposed to naive statelessness) in shims, and we explore some additional performance-oriented optimizations.

We use Intel Pin [LCM+05], an x86-based dynamic binary translation framework, to implement our software shims. Pin is a widely-used tool which allows users to write custom "pintools", or instrumentation routines, to perform the desired analysis. Because Pin executes on the x86 architecture and therefore has TSO as the downstream model, we use SC as the upstream model.

We evaluate three shim configurations. The first is the *naive* case which always inserts a LOCKed instruction or `mfence` between each pair of memory instructions. The second is the *stateful* shim generated by Algorithm 4.1 and shown in Figure 4.2. Third, the *ISA-assisted* scenario approximates the benefits of augmenting an ISA to track software- or compiler-provided information about accesses that do not need to enforce consistency. An increasing body of work has proven the benefits of providing hardware support for finer-grained specification of memory consistency behavior [CKS+11, SNM+12]. Because we are constrained by Pin's need to execute on real hardware (which has no such ISA support), we instead present approximations which closely model the performance benefits of enabling such modifications.

The ISA-assisted scenario considers two ways in which the ISA can be augmented. First, certain accesses might be marked thread-private and hence not subject to reordering rules. Even relatively straightforward compiler analysis is able to classify as many as 81% of memory accesses [SNM+12] as private. We approximate this by inferring thread-privacy for all accesses to the stack. While this is not safe in general, our analysis reveals that it is safe for our benchmark suite[3]. This approximation

[3]There are cases in which worker threads access objects allocated by the main thread, but these are properly synchronized via `pthreads`.

classifies 75% of accesses as thread-private, very close to the percentage found by the previous work.

Second, we model the benefits of a compiler annotating memory accesses as being data-race-free, and thus not subject to any reordering constraints [AH90, BA08]. For our pre-C11/C++11 benchmark suite, all synchronization accesses occurred through libraries such as `libpthread` or inline assembly, with the remainder of the program accesses remaining data-race-free. Because library behavior may not be precisely known at compilation time, we conservatively assume that all library code is potentially subject to races (and hence needs to be analyzed by the shim).

We run Pintool experiments on the real system from Table 4.1. We use benchmarks from PARSEC [Bie11] with the native input set and four threads. We take three measurements for each scenario: the non-Pintool native runtime of the benchmark ("native"), the runtime of the benchmark with analysis enabled but fence insertion itself disabled ("instrumentation"), and the runtime with fence insertion enabled ("shim"). This allows us to roughly separate the overhead of the shim from the overheads of Pin itself. We use LOCK-prefixed `add` instructions as the primary downstream MOR; these are equivalent to `mfence` but 28% faster in our experiments.

4.4.2 Hardware Simulation Methodology

While Pin offers opportunities for early exploration, hardware support can further accelerate translation. Here we study incorporating shims as hardware FSMs within a processor pipeline. In particular, we use the gem5 simulator to implement a hardware shim within the issue queue of the gem5 O3 pipeline [BBB+11]. In the issue queue, the shim has enough information to properly track both instruction dependencies and preserved program order. It does not, however, have enough information to distinguish same- vs. different-address relationships, and so the experiments in this section always conservatively assume that both need to be enforced.

	PO+ SA Ld	PO+ DA Ld	PO+ SA St	PO+ DA St
PO Ld	✓	✓	✓	✓
PO St	✓	✓	$\checkmark_S$	$\checkmark_S$

(a) Full fence

	PO+ SA Ld	PO+ DA Ld	PO+ SA St	PO+ DA St
PO Ld	✓	—	✓	✓
PO St	$\checkmark_L$	—	$\checkmark_M$	$\checkmark_M$

(b) PLO PPO

	PO+ SA Ld	PO+ DA Ld	PO+ SA St	PO+ DA St
PO Ld	✓	—	✓	✓
PO St	$\checkmark_L$	—	$\checkmark_S$	$\checkmark_S$

(c) MSFence

	PO+ SA Ld	PO+ DA Ld	PO+ SA St	PO+ DA St
PO Ld	✓	✓	✓	✓
PO St	$\checkmark_L$	—	$\checkmark_M$	—

(d) PSO PPO

	PO+ SA Ld	PO+ DA Ld	PO+ SA St	PO+ DA St
PO Ld	✓	✓	✓	✓
PO St	✓	✓	$\checkmark_M$	—

(e) MLFence

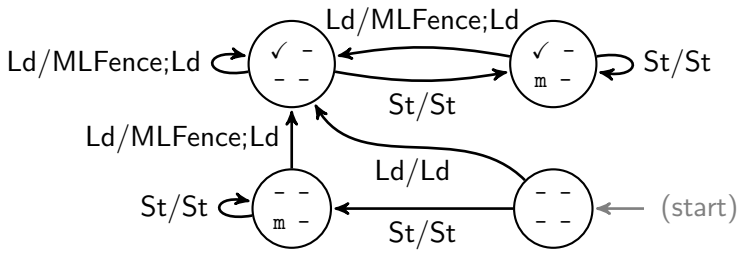
	PO+ SA Ld	PO+ DA Ld	PO+ SA St	PO+ DA St
PO Ld	✓	—	✓	✓
PO St	$\checkmark_L$	—	$\checkmark_M$	—

(f) LSO PPO

Figure 4.6: Available downstream PPO and MORs of the gem5 O3 simulated CPU.

By design, gem5 decouples the pipeline model itself from the ISA being implemented, and so the same pipeline can be used for each scenario. This fact makes gem5 a good simulation environment for isolating the effect of memory consistency models from the effects of the ISA itself. As Figure 4.6 summarizes, the gem5 O3 pipeline is multi-copy-atomic and inherently enforces load→store ordering, while load→load and/or store→store ordering enforcement (of strength $\checkmark_S$) are optional. We adjust these options to implement a variety of downstream models.

At the gem5 O3 issue queue, regardless of the architecturally-defined fences, three downstream fences are available microarchitecturally: a load fence, a store fence, and a full fence. The MOSTs for these fences are also summarized in Figure 4.6. The fences are implemented microarchitecturally by treating an associated memory access as non-speculative. This requires that before the access executes, it must be at the head of the reorder buffer and the store buffer must be empty. The associated



MLFence

	PO Ld	PO St
PO Ld	✓	✓
PO St	✓	—

Figure 4.7: The shim FSM generated for SC on PLO (for visual clarity, the MOST and pending ordering tables are abbreviated). The bottom two states are transient because there are no downward arrows to return to them from the top two states. The top two states are redundant because their behavior is identical. Therefore, this FSM reduces to a single state.

operation is also treated as a load and/or store barrier to prevent subsequent memory microops of the relevant type(s) from executing until it has itself completed.

Table 4.1 gives specifications for our simulated system. Because the generated FSMs are small, we assume they can be updated in parallel with other pipeline operations with no incurred latency. We use PARSEC [Bie11] benchmarks with the simsmall input set and 4 threads. We execute these benchmarks on four downstreams: TSO, PSO, PLO (partial load order, named by analogy to SPARC PSO), and LSO (load→store order enforced). We compare the performance for each case, including both stateless and stateful shims.

4.5 Experimental Results

4.5.1 Shims for a Broad Range of Scenarios

To demonstrate the full breadth of applicability of ArMOR, we automatically generate shim FSMs for various combinations of the upstream and downstream consistency models. A summary is shown in Table 4.2; the shaded subset highlights the scenarios evaluated in the next two sections. The full analysis is presented in Appendix B. We also optimize away transient and redundant states, as in Figure 4.7, to reduce the

Model	Informal Description
SC	Sequential consistency
TSO	SPARC Total Store Ordering
PSO	SPARC Partial Store Ordering
PLO	Partial load ordering: TSO, except load→load ordering is not enforced
LSO	Only load→store ordering is enforced
RMO	SPARC relaxed memory ordering
RMO+	RMO variant with sixteen fences representing four independent choices of load→load, load→store, store→load, and store→store ordering
PwrA	A multi-copy atomic variant of Power
Pwr	Power
ARM	ARMv7

(a) Memory models evaluated (see Appendix B for details)

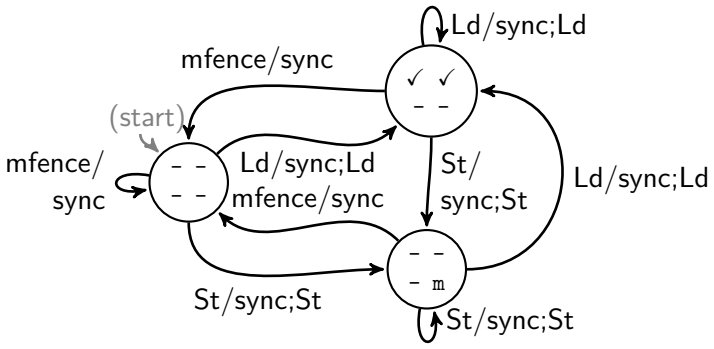
	Target MCM								
Source MCM	**TSO**	**PLO**	**PSO**	**LSO**	**RMO**	**RMO+**	**PwrA**	**Pwr**	**ARM**
SC	2	1	2	1	1	2	2	1	1
TSO	-	2	2	4	3	5	4	1	1
PLO	-	-	2	2	2	4	3	1	1
PSO	-	2	-	2	3	3	2	1	1
LSO	-	-	-	-	2	2	2	1	1
RMO	-	-	-	-	-	-	1	1	1
PwrA	-	-	-	-	-	-	-	1	1
Pwr	-	-	-	-	-	-	-	-	1
ARM	-	-	-	-	-	-	-	1	-

(b) Number of states in shim for each pairing of upstream and downstream models. The performance of the shaded FSMs is evaluated in Section 4.5.3.

Table 4.2: ArMOR analysis can be used to generate shims for a broad range of upstream and downstream memory model pairings

implementation cost. These results show that shim FSMs are generally very small, and hence that they can be implemented in practice with little area cost.

Notably, adding state does not help in every situation. For example, Figure 4.8 shows that the FSM generated for TSO upstream and ARM/Power downstream minimizes to a trivial machine which always inserts `sync`. While it may seem to be overkill to insert a `sync` before each memory access, Figure 3.2 highlighted that anything weaker would permit behaviors such as `iriw` to become illegally observable. In fact, every multi-copy-atomic upstream paired with a downstream allowing non-multi-copy-atomic stores produces a FSM which is just as inefficient. This is not a



sync

	PO Ld	PO St
PO Ld	✓	✓
PO St	✓	✓

Figure 4.8: Automatically-generated shim FSM for TSO upstream and Power downstream (for visual clarity, the MOST and pending ordering tables are abbreviated). This FSM can be simplified into a single state.

shortcoming of ArMOR, but rather a fundamental difference in the behavior of stores on each architecture. We return to this observation in Section 4.6.

Lastly, we note that we attempted to target GPUs as well, but we were limited both by the incompleteness of current specifications as well as, more fundamentally, a lack of both multi-copy atomicity and cumulative fences [ABD+15]. With neither feature, GPUs simply have no means by which to enforce implicit (e.g., on TSO) or explicit (e.g., Power `sync` or ARM `dmb`) cumulativity requirements, and hence they are unable to serve as downstream targets for translation.

4.5.2 Performance of Software Shim Implementations

Figure 4.9 shows the performance of the three Pintool shim configurations of Section 4.4.1. We normalize to the runtime of each benchmark when it is compiled for x86-TSO and executed natively on x86-TSO hardware; this conservatively attributes the inherent overhead of SC vs. TSO to shims as well. Ideally, an optimized "native x86-SC" machine would be a more appropriate baseline, but such hardware is not readily available.

Each level of optimization provides dramatic performance increases over more naive configurations. The stateless shim has a geometric mean overall performance

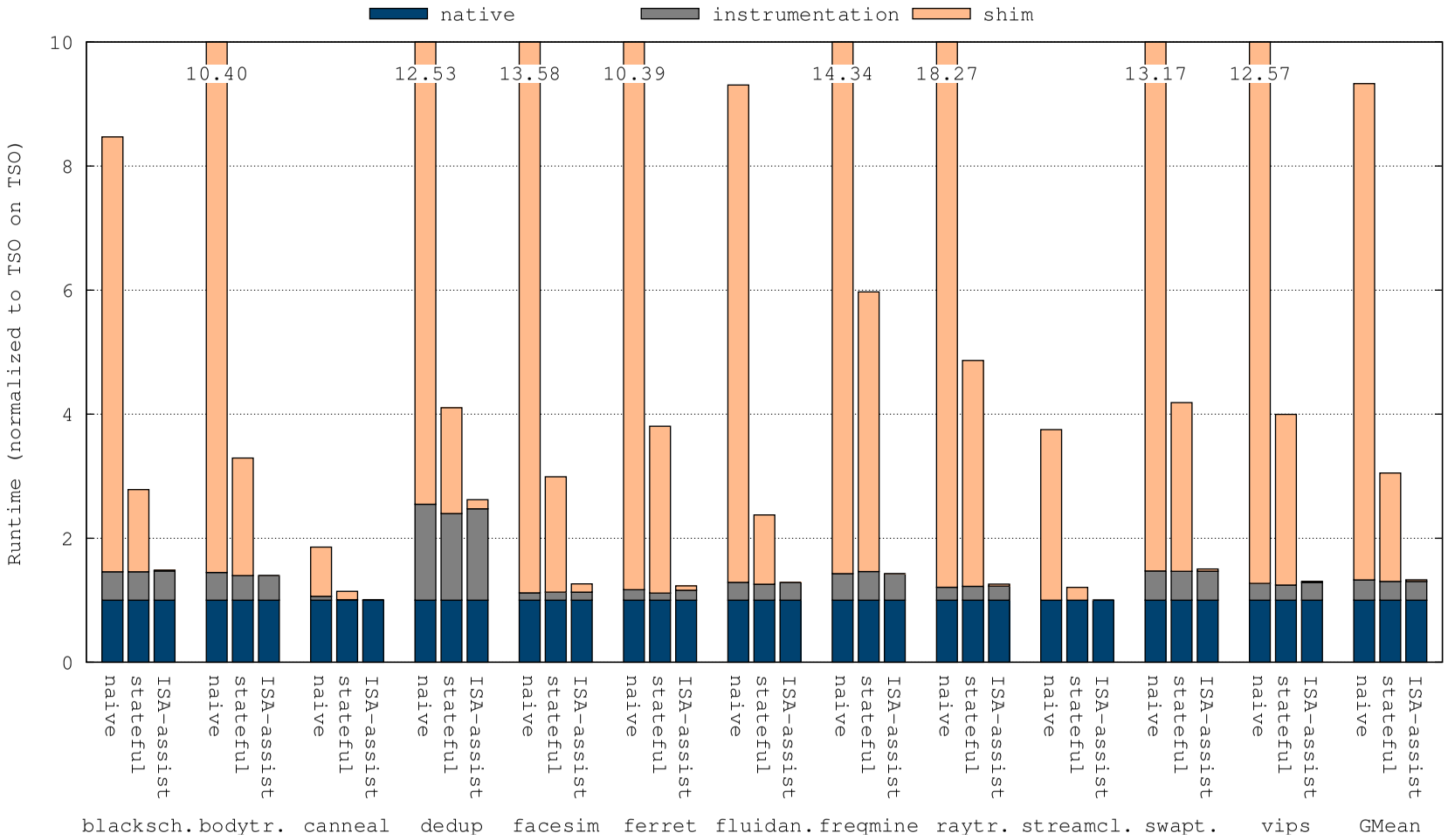


Figure 4.9: Performance overhead of shims using dynamic binary translation and different levels of performance optimization. From left to right, the three bars represent the stateless, stateful, and ISA-assisted stateful cases, respectively.

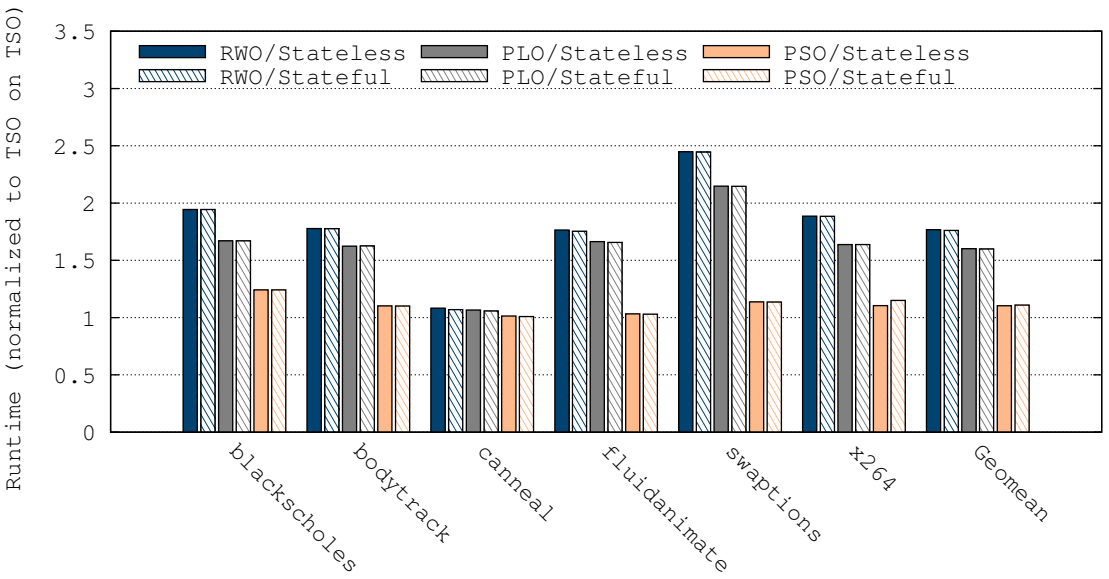


Figure 4.10: Simulated performance with hardware shims, x86-TSO software and varying hardware models.

cost of 9.33×. The stateful configuration improves this to 3.05×. Finally, making use of the ISA augmentations discussed in Section 4.4.1 reduces the total overhead to just 1.33×.

The instrumentation overhead was approximately the same for each Pintool—1.31× on average. This shows that the shims themselves do not introduce significant overhead beyond the overhead of instrumentation itself—175% in the case of our conservative stateful configuration, but only 3% in the more aggressive ISA-assisted case. These numbers demonstrate that shim-based translation can take place with low or, under the right conditions, even negligible overhead in practice beyond what is already needed to perform dynamic binary translation. They also demonstrate the value of using software-based DBT as a tool for exploring the design space of and profiling the use of synchronization in practice.

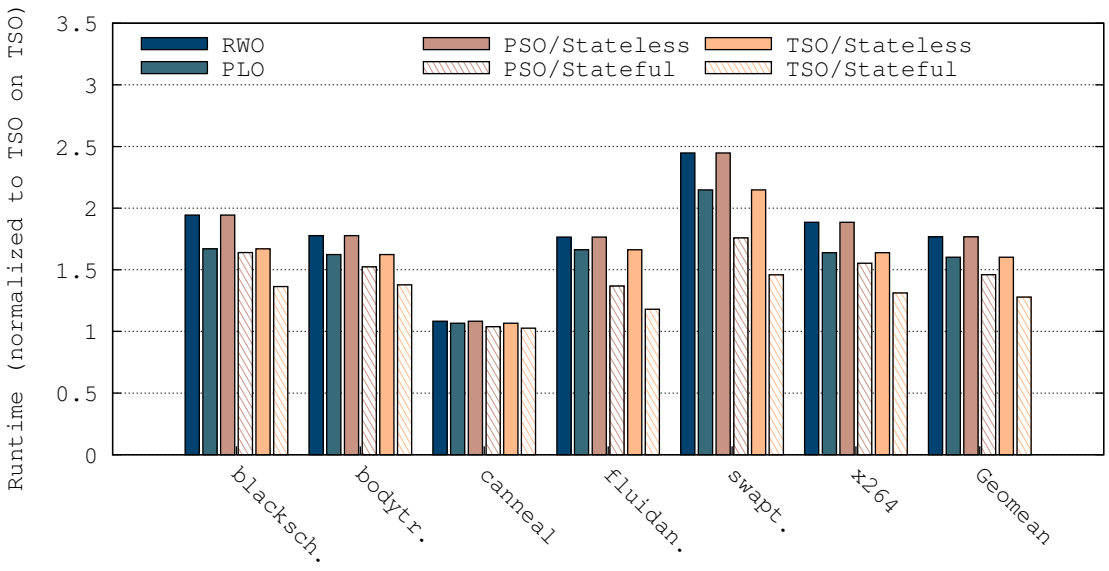


Figure 4.11: Simulated performance with hardware shims, x86-SC software and varying hardware models, normalized to x86-TSO software on x86-TSO hardware as described in the text.

4.5.3 Performance of Hardware Shim Implementations

Figures 4.10 and 4.11 show the overheads of running x86-TSO and x86-SC software, respectively, on cores with weaker hardware memory models. Stateful shims are shown with striped bars; however, the optimized FSMs for SC-on-PLO and SC-on-LSO are stateless, and so we draw only one bar for these. We normalize to x86-TSO hardware, just as we do for the Pintool results.

For x86-TSO upstream, in some cases, the benefits of statefulness are negligible. Since upstream LOCKed operations and fences are rare under x86-TSO, these FSMs turn out to mostly stay in a single state equivalent to the stateless case. For this reason, implementations may choose to treat the lightly-used state as transient (as in Figure 4.7) and optimize it away to make the FSM even cheaper than it already is.

Notably, if the shims had been placed in a position at which they could distinguish and track whether acceses were made to the same or to different addresses, then the distinction between stateful and stateless shims would become more meaningful. However, this positioning and/or tracking may itself introduce new area, power, or

performance overheads which may outweigh the benefits of the more sophisticated shims. These kinds of design decisions must be made on a case-by-case basis. Fortunately, ArMOR provides the flexibility to derive shim designs for any such case.

In all cases, the overheads remain well within an acceptable range for dynamic binary translation [BCC+10, LCM+05]: even the worst case overhead of SC on RMO requires a geomean slowdown of only 77%. In the best case, TSO on PSO, the overheads are as small as 10%.

4.6 Takeaways

Our explorations both via Pintools and via simulation of hardware-supported shims have led to several major takeaways. First, *architectures should provide a way to optionally make stores multi-copy atomic.* A multi-copy atomic variant of Power, labeled "PwrA" in Table 4.2, does allow for more efficient FSMs than does the original non-multi-copy-atomic version. If the user is sure that no `iriw`-like behavior will occur, then multi-copy atomicity can be disabled to improve performance; otherwise, it can be enabled to ensure safety[4]. Notably, ARMv8 has taken this approach with new load-acquire and store-release opcodes [ARM13]. ArMOR provides a rigorous methodology for performing this analysis.

Our second observation is that *the more downstream MORs (i.e., fence variations) are available, the more intelligent the translation can be.* The difference between "RMO" and "RMO+" in Table 4.2 is that the former implements only the three fences shown in Figure 4.6, while the latter implements sixteen possibilities, with one choice each for load→load, load→store, store→load, and store→store. Having finer-

[4]The `iriw` litmus test is frequently debated in the memory model community; it is generally considered esoteric and unlikely to occur in reality, and the cost of preventing it is generally large, yet the behavior is highly counterintuitive and as a result it is often forbidden in spite of the cost [Alg12, BA08].

grained downstream fences allows for smarter fence choices and higher-performance implementations.

Third, *ISAs and intermediate representations should maintain consistency metadata even if it is redundant.* In particular, ISAs with strong models (e.g., SC, TSO) carry little information about consistency in the binary, as it is mostly redundant with PPO. However, this lack of metadata makes translation much more difficult, as the overly-constrained preserved program orderings of a strong model like TSO are themselves costly and mostly unnecessary. Specifically, weak software models derive their performance from enforcing strong ordering only on specially-marked synchronization accesses. On strong hardware models in which both types of accesses map onto the same loads and stores, non-synchronization accesses can no longer be reordered freely, even when translation back onto a weaker model, as the metadata has by that point been lost. Keeping consistency information in the ISA would provide numerous benefits (shown in Section 4.5.2 and previous work) at the cost of modest code size increase. In such a scenario, shims can be used to dynamically remove upstream synchronization that becomes redundant under a stronger downstream model.

Finally, we note that *non-multi-copy-atomic architectures cannot ignore both cumulativity and store atomicity.* If they do, then there simply is no way to implement communication across more than two cores safely. While current hardware that does so (e.g., GPUs) simply limits the amount of inter-thread communication that can take place, the increasingly heterogeneous hardware of the future will demand the ability to perform such many-threaded concurrent tasks. Fortunately, ArMOR provides a way to evaluate those needs early in the design process.

4.7 Related Work

Fence Insertion and/or Elimination. The work of Alglave et al. [AMSS10] has a goal similar to ours in that it studies how to restore the behavior of one architecture by inserting fences on a weaker architecture. Their definition of cumulativity is subtly different than the definition given in the Power architectural specification [IBM13], and their proof-based method does not readily adapt to a modified definition. More critically, their solution is declarative: it specifies only a static correctness condition rather than a constructive dynamic translation method. Furthermore, their correctness condition depends partially on inserting fences between loads and their source stores. ArMOR makes no such assumption about identifying a load's source store, as such information is often simply unavailable.

Since the work of Shasha and Snir [SS88], researchers have considered topics such as verifying the insertion of fences to implement a stronger consistency model [BAM07, KVY10] and/or the elimination of redundant fences [VN11]. Others focus on automatically determining where to insert fences [Alg12, HR07], and also on incorporating such methods into a compiler [LP00, SFW$^+$05].

Cross-ISA Translation. DeVuyst et al. [DVT12] study heterogeneous-ISA code migration. They focus on laying out data in an architecture-independent manner, and they use compiler support and bursts of dynamic binary translation to smooth the migration process. They assume, however, that the source and target ISAs have identical consistency models; they do not address translation of memory ordering requirements.

Various case studies have studied translation in more specific contexts, including Baraz et al. [BDE$^+$03] for x86 code on Itanium processors, Higham and Jackson [HJ06] for SPARC to and from Itanium, and Gschwind et al. [GEAS00] from the "firm" model (similar to TSO) onto Power. Industry white papers [Bro02] have also discussed this

topic. None of these techniques, however, easily generalize to other architectures as ArMOR does.

4.8 Chapter Summary

This section provided an in-depth case study of the broad applicability of MOSTs. It demonstrated not only how ArMOR analysis techniques can be applied to various existing architectures, but also how they can be applied to varying microarchitectures and/or to hypothetical ISA extensions as well. Furthermore, it showed how ArMOR can be used to deliver both correctness and performance in use cases which are forward-looking and which will likely become increasingly relevant. In summary, we believe that ArMOR can inspire the designs of future heterogeneity-aware architectures and that it can be easily extended to support many other MCM analysis use cases in the future.

Appendix B presents a gallery of dynamic binary translation shims for a wide range of upstream and downstream memory model combinations.

Chapter 5

PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models

Previous chapters have focused on developing a specification language and an analysis framework for memory models. This chapter and the next instead focus on specifying and verifying how a given memory model specification is actually implemented at the microarchitecture level.

5.1 Introduction

Memory model analysis performed at the architecture level—that is, at the level of the specification provided by vendors—intentionally abstracts away the details of any one particular microarchitecture itself. This fact leaves such models unable to describe how the ordering requirements of the memory model are actually enforced.

In particular, although any one particular microarchitecture may implement optimizations such as out-of-order execution or speculative reordering of instructions, architecture-level memory models are by design unable to capture such behaviors or to check whether they correctly enforce all required orderings. This leaves a large and problematic verification gap between the careful formal analysis happening at the software and architecture levels and the reality of how high-performance or even commodity processors implement those models at the microarchitecture level. As a result, implementations of memory system features frequently contain bugs which can only be solved through high-overhead patches [AMD11, ARM11] or by simply disabling new features altogether [Int15].

This chapter presents PipeCheck, a tool and methodology for specifying microarchitecture-level multi-event axiomatic consistency models and for automatically verifying that those models meet (or exceed) the requirements of a given architectural consistency model. Each microarchitecture-level model consists of a set of localized axioms about the behavior of the microarchitecture at various physical locations. The relationships prescribed by these axioms are then merged to form *microarchitectural happens before* (μhb) graphs, an extension of standard happens-before graphs used in axiomatic memory models to describe microarchitecture-level events and happens-before relationships. Using these graphs, PipeCheck verifies that each ordering edge that must be preserved according to the architectural consistency model (e.g., each store→store ordering for Total Store Ordering (TSO)) is in fact provably maintained by the microarchitecture. As a result, PipeCheck reduces the problem of verifying consistency model implementation correctness to the more tractable problem of verifying a series of independent axioms about ordering enforcement at various points in the microarchitecture. It also reinterprets abstract architecture-level axioms such as preserved program order (Section 2.2.1) as propo-

sitions which can themselves be re-derived from the same microarchitecture-level axioms.

At the microarchitecture level, consistency is collectively enforced by the combination of many components: the pipeline, the cache hierarchy, the coherence protocol, and so on. Furthermore, while properties such as coherence and consistency are often intentionally decoupled at the architecture level [Mar05], they are nevertheless often tightly coupled at the microarchitecture level. This thesis decomposes the problem into two chapters. This chapter presents the PipeCheck approach and μhb graphs in general, and it focuses on the enforcement responsibilities of the pipeline under the assumption of an "idealized" memory system. Chapter 6 then analyzes in depth the responsibilities of various coherence protocols in enforcing consistency, and it then delivers a complete model of their combined enforcement.

The rest of this chapter is organized as follows. Section 5.2 describes a motivating example. Section 5.3 describes the PipeCheck microarchitecture-level approach. Section 5.4 presents the verification approach. Our analysis methodology and tool flow is described in Section 5.5. Sections 5.6 and 5.7 summarize overall results and highlight important case studies, respectively. Lastly, Section 5.8 describes related work, and Section 5.9 concludes. Finally, Chapter 6 extends PipeCheck analysis to coherence protocols and to the coherence-consistency interface.

5.2 Motivating Example

Recall from Chapter 2 that axiomatic memory model analysis typically involves the creation of a happens-before (hb) graph subject to certain axioms or constraints. Happens-before graphs represent each program as a graph in which vertices represent memory instructions and/or abstracted instruction visibility events in the program. Edges between these vertices represent ordering relationships between the source and

destination of the edge: an edge from an instruction s to another instruction d indicates that s *happens before* d, in some formal sense defined by the model. Figure 5.1 presents an example using litmus test `mp`. The fact that analyzing the litmus test (Figure 5.1a) according to the TSO specification (Figure 5.1b) results in a cyclic graph (Figure 5.1c) indicates that the result is forbidden.

The specification of TSO in Figure 5.1b uses the following edge types which are (informally) summarized below [Alg12]:

- "reads from" (`rf`): if d reads from s, then s must happen before d, from at least some points of view. "rfe" (`rf`-external) represents `rf` orderings between different cores.

- "write serialization" or "coherence order" (`ws`): s comes before d in the (assumed) total ordering of all stores accessing a given location, from the point of view of the memory hierarchy

- "from reads" or "reads before" (`fr`): s is a read that gets its value from a write that comes before d in the set of `ws` edges. "fre" (`fr`-external) represents `fr` orderings between different cores.

- "mfence" (`mfence`): s comes before d in program order, and there is an `mfence` instruction in between them in program order

- "preserved program order" (`ppo`): s comes before d in program order, and either s is a load or d is a store

- "program order, same location/address" (`po-loc`): s comes before d in program order, and s and d access the same memory address

One goal of PipeCheck microarchitecture-level analysis is to reinterpret the abstract architecture-level relationships listed above in the context of a particular microarchitecture. Some of these abstract edges will be made more concrete, in the

Thread 0	Thread 1
(i1) [x]←1	(i3) r1←[y]
(i2) [y]←1	(i4) r2←[x]

Outcome `r1=1, r2=0`:
Forbidden under TSO

(a) Litmus Test Code

$$\text{acyclic}\,(\texttt{rfe} \cup \texttt{ws} \cup \texttt{fre}^{1} \cup \texttt{ppo} \cup \texttt{mfence})$$

$$\text{acyclic}\,(\texttt{rf} \cup \texttt{ws} \cup \texttt{fr} \cup \texttt{po-loc})$$

(b) Axiomatic specification of the TSO memory model [Alg12]

(c) The cycle in the graph for the first axiom indicates that this execution is forbidden.

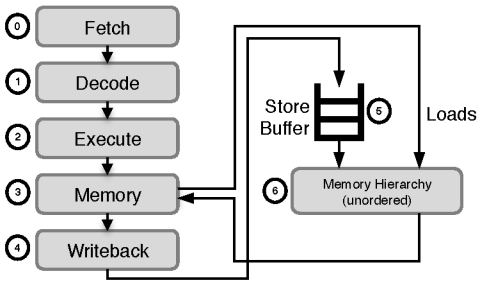
Figure 5.1: Load→load and store→store ordering litmus test `iwp2.1/amd1/mp`.

sense that they will become associated not just with instructions but also with specific points in a pipeline. Others will be entirely replaced by multiple nodes and edges that represent finer-grained relationships that correspond to specific microarchitecture-level behavior, as discussed below.

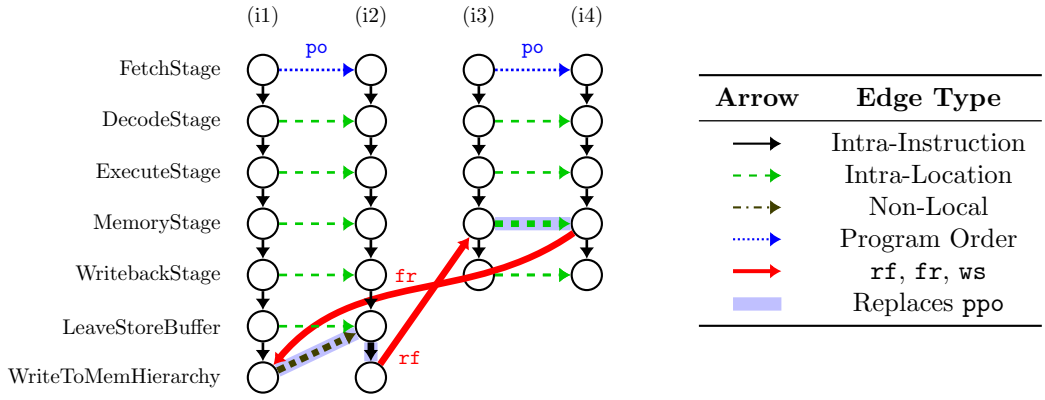
5.2.1 Microarchitecture-Level Analysis

The analysis in Figure 5.1c says nothing about the behavior of any individual *microarchitectural implementation* of that architecture. On one hand, certain architecturally-permitted behaviors may not be observable on a given microarchitecture. For example, a sequentially consistent (SC) pipeline is a valid implementation of a TSO architecture, although many executions that are legal under TSO will not be observable in such a pipeline—the microarchitectural memory model is stricter than the architecture requires. On the other hand, architecturally-forbidden behaviors may be observable on a given microarchitecture—this would correspond to a bug in the

[1] Alglave et al. use `fr` in place of `fre` here, but the two are equivalent: `fr`-internal, `fri`=`fr`−`fre`, must be in agreement with `po` by the second axiom, and therefore `fri` is contained in `ppo`.



(a) Classic five stage RISC Pipeline plus a FIFO store buffer and an unordered memory system. This pipeline example recurs throughout this chapter.



(b) In this example, the preserved program order (`ppo`) edges are discarded as an architecture-level assumption, and the model verifies that enforcement required by those edges is instead enforced by some combination of μhb edges.

Figure 5.2: Microarchitecture-level analysis of `mp` on the given microarchitecture

implementation. In this case, the microarchitecture is erroneously weaker than the architecture requires.

As a running example of a microarchitecture, we will use the classic RISC five-stage pipeline, augmented with a store buffer, as shown in Figure 5.2a. The meanings of the nodes and edges are described in detail in Section 5.3.2. For clarity, we assume here that this microarchitecture has no cache; Chapter 6 will describe how to model cache behavior in pipelines for which consistency and coherence are interdependent. Furthermore, in this example we assume that the memory system and any interconnects and networks-on-chip are unordered. This assumption is made in order to highlight an important point: even for an in-order pipeline, the presence of an unordered network and/or memory hierarchy can result in memory accesses being re-

ordered from the point of view of other processors. As a result, PipeCheck must also account for such behavior. Although we use a simple example here to build intuition, we verify more complex processors, including OpenSPARC T2, later in the chapter.

Figure 5.2b shows the edges from Figure 5.1c as translated into the PipeCheck model of the microarchitecture of Figure 5.2a. The four memory operations, (i1), (i2), (i3), and (i4), are depicted from left to right, and various stages in the microarchitecture are shown from top to bottom. Specifically, each vertex corresponds not just to a memory instruction, but also to a particular *location* within the pipeline or memory system. Each column of vertices therefore corresponds to that instruction progressing through the various locations in the microarchitecture. The various edge types will be described in detail in later sections.

While `ws` edges are defined with respect to main memory, PipeCheck maps the endpoints of `rf` and `fr` edges to the points in the pipeline at which the instructions can be considered to have *performed* with respect to the core at the other endpoint. As discussed in Section 2.1.1, a store from core i has traditionally been said to have performed with respect to core j when a load of the same address issued by core j returns the value written by the store (or by some subsequent store) [DSB86, SD87]. A load from core i has performed with respect to core j when a subsequent store from core j to the same address cannot affect the value returned by the load. A store or a load has *performed globally* when it has performed with respect to all cores. More recent studies, however, have criticized this definition as being too hypothetical for formal analysis [AFI+09, NSS+09].

From a microarchitectural point of view, Section 5.3.1 will therefore unambiguously define "perform" in terms of *locations* in the pipeline. Continuing the example, the `rf` edge from (i2.WriteToMemHierarchy) to (i3.MemoryStage) indicates that instruction (i2) must have performed with respect to core 1 (i.e., been written back

from core 0 to the memory hierarchy) before instruction (i3) performs with respect to core 0 (i.e., reads memory from the memory stage of the core 1 pipeline).

5.2.2 Deriving Correctness from Microarchitecture-Level Axioms

A key benefit of our microarchitecture-level analysis is that it treats microarchitecture-level correctness as a goal that can be derived from a set of simpler, easier-to-verify self-contained *axioms* of a microarchitecture. Each microarchitecture-level axiom is independent, and hence each can be independently verified as truly representative of the underlying transistors. The goal of this approach is to create a situation in which verification of the validity of each individual axiom with respect to the register transfer level (RTL) specification of the microarchitecture is significantly more tractable than verification of the microarchitecture as a whole would be.

Figure 5.2b presents one example of the axiom-based approach. In this figure, we draw (in dashed green) orderings that are maintained at each location individually; this calculation will be the focus of Section 5.3. In this example, with an in-order pipeline, each pipeline stage independently maintains at its output the instruction ordering it observes at its input. Taken together, these axioms result in the `po` orderings being maintained throughout the pipeline, with the notable exception of writes sent to memory. The ordering of stores at the memory hierarchy is instead maintained by the store buffer via the diagonal edge from (i1.WriteToMemoryHierarchy) to (i2.LeaveStoreBuffer).

The analysis in Figure 5.2b treats `ppo` edges as *propositions* rather than axioms: it checks that the pipeline properly maintains the `ppo` edges, rather than simply assuming their presence. Consider the `ppo` edges that were part of the cycle in Figure 5.1c. While their presence was assumed in the architecture-level analysis, PipeCheck does not assume them; it *checks* for them. The `ppo` edge from (i1) to (i2)

is enforced by a sequence of three microarchitecture-level "happens before" edges, as shown highlighted in gray in Figure 5.2b. The `ppo` edge from (i3) to (i4) is enforced by a single microarchitecture-level edge. Together with the observed microarchitecture-level `rf` and `fr`, the union of the highlighted edges forms the microarchitecture-level equivalent of the cycle in Figure 5.1c. Thus, at least for this particular example, the `ppo` edges are correctly maintained, and hence the pipeline correctly implements the TSO restrictions on this litmus test. While this example covers only one test, later sections will describe the process of fully verifying all possible orderings.

5.3 PipeCheck Microarchitecture-Level Analysis

A central observation of PipeCheck is that orderings between instructions are often too complicated to be captured by a single architecture-level "happens before" edge. A single pair of instructions may *fetch* in order, *issue* out of order, *execute* out of order, *commit* in order, and *perform globally* out of order. Restricting "happens before" edges to specify only orderings with respect to the memory hierarchy ignores all of the other important memory orderings that take place within the pipeline itself. PipeCheck therefore defines "microarchitectural happens before" (μhb) edges to specify both an instruction and a particular *location* within the pipeline:

Definition 1 (Microarchitectural Happens-Before (μhb))**.** *A μhb graph is a directed graph (V, E) in which each vertex $v \in V$ represents some microarchitecture-level event taking place at some particular physical location, and in which each edge $(s, d) \in E$ represents a guarantee that event s happens before event d.*

Most commonly, each microarchitecture-level event refers to some instruction i passing through some particular physical location l (e.g., a particular pipeline stage) in the microarchitecture. However, events can also refer to the sending or receiving of a message, a change in the state of some element, or any other physical event. The

requirement that each event be associated with some location is key to the transitivity properties that we discuss in Section 5.3.3.

Throughout this thesis, we depict μhb graphs in a grid, as in Figure 5.2b, with instructions along the x-axis and microarchitectural locations or events along the y-axis. Not all instructions pass through all locations (e.g., loads do not occupy the store buffer), and so some entries in the grid are left empty. Despite the grid depiction, only relationships depicted by arrows provide any ordering guarantee.

As with any model checking approach, PipeCheck is able to verify a pipeline only to the extent that the model provided is faithful to the orderings enforced by the underlying implementation. If the user specifies an axiom which is not in fact guaranteed by the microarchitecture, then PipeCheck may erroneously declare a pipeline to be correct, even though it may actually be buggy. On the other hand, if the user omits an ordering axiom which turns out to be valid, then PipeCheck may erroneously declare a pipeline incorrect, even though it may actually work correctly. A key advantage of the PipeCheck approach is that each PipeCheck axiom tends to be more localized and/or finer-grained, meaning that verification of each axiom individually should be significantly easier than verifying coarser, more abstract architecture-level edges. For example, the next subsection will decompose the coarse-grained preserved program order `ppo` edges into a series of finer-grained, per-pipeline-stage ordering guarantees. Since each per-stage axiom is localized to a single module of the pipeline, it will be easier to each per-stage property than to verify the correctness of the coarser `ppo`, which involves many pipeline components.

5.3.1 Microarchitecture Definition

In PipeCheck, a microarchitecture is modeled as a set of *microarchitecture-level axioms*. These axioms specify various properties of and/or orderings enforced by the given microarchitecture. For example, axioms may specify:

- The legal *path(s)* per instruction type.
- A *local ordering guarantee* at each pipeline stage
- *Performing locations* within each path, and a specification of how instructions interact with each other
- Edges specifying non-local happens-before relationships enforced via *sending/receiving of messages*
- Any other edges which represent orderings that are somehow guaranteed to be enforced

These terms are more carefully defined below.

During execution, as instructions flow through the pipeline, they pass through the chosen locations along some well-defined *path*. A memory instruction may have more than one legal path through a pipeline, depending on the type of instruction, the state of the pipeline, and/or the state of the memory system during execution. For example, a read may take a different path depending on whether it performs by reading from the store buffer, from the cache via a cache hit, or from the cache after a cache miss. Paths may overlap entirely, significantly, or not at all. A typical path will involve a straight-line sequence of μhb edges enforcing the order in which each instruction passes through pipeline stages: e.g., Fetch$\rightarrow$Decode$\rightarrow$ $\cdots$ $\rightarrow$Commit. Different pipelines will of course require varying sequences of path edges.

To more precisely define "in order" and "out of order" sections of a pipeline, we can define a *local reordering guarantee* axiom for each pipeline stage. This specifies the reorderings that the pipeline stage does or does not permit on instructions passing through it. In one extreme, a *FIFO* local reordering guarantees that instructions arriving at a particular pipeline stage in a certain order will leave that stage in the same order. For example, a local reordering guarantee for a pipeline decode stage

may make the following guarantee:

$$\text{If: } (i1, \text{Fetch}) \xrightarrow{\mu hb} (i2, \text{Fetch}),$$
$$\text{Then: } (i1, \text{Decode}) \xrightarrow{\mu hb} (i2, \text{Decode}).$$

At the other extreme, a pipeline stage may not guarantee any such orderings, and so there would be no local reordering guarantee axiom for such locations. For example, an out-of-order issue stage or an unordered network may fall into this category. In between the two extremes can be any other form of guarantee, such as "maintain orderings between dependent instructions only" at an issue queue. Also, notably, the local reordering performed by a reorder buffer stage may take the form of "restore the ordering seen at the entrance to the rename stage", for example. The specific guarantees of each pipeline stage will vary from processor to processor.

Inter-instruction μhb edges specify how instructions interact with one another. At a high level, these edges correspond to static relationships (e.g., address, control, and/or data dependencies) and/or dynamic dependencies (e.g., reads-from, write serialization, or from-reads). Each such edge cannot just be interpreted as a purely inter-instruction relationship, however. As each μhb edge specifies a relationship between particular μhb nodes, each inter-instruction μhb relationship inherently specifies an ordering between two events at two specific physical locations. In this way, μhb edges are more directly representative of the actual microarchitecture-level mechanisms used to enforce the specified ordering than more abstract architecture-level edges such as preserved program order (`ppo`).

For example, consider a reads-from (`rf`) relationship. If a load reads from a previous store on the same pipeline, then one μhb interpretation is to say that the store must have entered but not yet left the store buffer at the time the load accesses

it. This would imply two μhb edges:

$$(i1, \text{EnterStoreBuffer}) \xrightarrow{\mu hb} (i2, \text{AccessMemory})$$
$$(i2, \text{AccessMemory}) \xrightarrow{\mu hb} (i1, \text{ExitStoreBuffer}),$$

where EnterStoreBuffer and AccessMemory both occur in the MemoryStage in our running example. On the other hand, if a load reads from main memory, that store must have already been written into the memory hierarchy, in which case the following edge would be added:

$$(i1, \text{WriteToMemHierarchy}) \xrightarrow{\mu hb} (i2, \text{AccessMemory}).$$

The choice of source and destination nodes for each inter-instruction edge represents the *locations* at which each instruction can be said to have *performed* with respect to the target core. As described in Section 5.2.1, previous work defined the point at which an instruction performs as the point at which the effect of that instruction becomes visible to (possibly hypothetical) memory accesses from other cores [DSB86]. PipeCheck instead defines the point at which an instruction has performed with respect to a given core in terms of that instruction reaching some particular location within the microarchitecture. This maintains the intention of the traditional definition given in Section 5.2.1 while removing its hypothetical nature.

5.3.2 PipeCheck Model Specification Language

PipeCheck microarchitecture models are described using a μhb graph-centric domain-specific language (DSL) based on first-order logic[2]. Variables in the DSL represent instructions or threads. As shown in Table 5.1, the logic includes a pre-defined set of

[2]The original PipeCheck paper used a more primitive specification format directly embedded within the Coq specification language Gallina [LPM14, LPM15]. The DSL described in this thesis is a standalone evolution of that original approach.

Predicate	Argument Types	Notes
`IsAnyRead`	uop	
`IsAnyWrite`	uop	
`IsAnyFence`	uop	
`AccessType`	string, uop	e.g., `AccessType("RMW", i)` or `AccessType("lwsync", f)`
`SameMicroop`	uop, uop	
`SameThread`	uop, uop	
`ProgramOrder`	uop, uop	
`ConsecutiveMicroops`	uop, uop	
`InThread`	threadID, uop	
`SameThreadID`	threadID, threadID	
`SameAddress`	uop, uop	
`SameData`	uop, uop	
`DataFromInitialState`	uop	by address
`DataFromFinalState`	uop	by address
`NodeExists`	node	
`NodesExist`	list node	
`EdgeExists`	edge	
`EdgesExist`	list edge	

Table 5.1: Predicates used in the PipeCheck DSL

predicates ranging over variables of the types listed above. The predicates are used to build first-order logic formulas which represent the axioms of the model. Two predicates in particular—`NodesExist` and `EdgesExist`—are interpreted as adding (or, if they are negated, forbidding the addition of) nodes and edges to a μhb graph, respectively. In this way, the DSL encodes a primitive satisfiability modulo theories (SMT) problem over the theory of directed acyclic graphs [BSST09]. The constraint solver must do more than find a satisfying assignment for all of the literals; it must find a solution such that the nodes and edges represented by the predicates chosen form an acyclic μhb graph.

Figures 5.3 through 5.5 shows the PipeCheck definition of the classic RISC pipeline of Figure 5.2a. First, Figure 5.3 defines a set of macros that are expanded within the axioms. The macros serve as an organizational tool to keep the specification clean. Figure 5.4 then defines a set of axioms specifying the behavior of load, store and `mfence` instructions. Figure 5.5 adds three additional components. First, it adds

```
% Legend:
% "/\" = AND
% "\/" = OR
% "~"  = NOT
% "=>" = IMPLIES
% "%"  = COMMENT
%
% Graph node = (instruction, location/event)
% - The notation (instruction, (0, location/event)) indicates that all cores
%   share a single memory hierarchy, and that the shared memory hierarchy
%   is associated with core 0 in PipeCheck
% Graph edge = (node, node, label)
%
% "c" is predefined to be the core ID

StageName 0 "Fetch".
StageName 1 "Decode".
StageName 2 "Execute".
StageName 3 "MemoryStage".
StageName 4 "Writeback".
StageName 5 "LeaveStoreBuffer".
StageName 6 "WriteToMemHierarchy".

DefineMacro "STBFwd":
  exists microop "w", (
    IsAnyWrite w /\ SameCore w i /\ SameAddress w i /\ SameData w i /\
    EdgesExist [((w, MemoryStage), (i,     MemoryStage         ), "rf_stbuf");
                ((i, MemoryStage), (w, (0, WriteToMemHierarchy)), "rf_stbuf")]) /\
    (~exists microop "w'",
      IsAnyWrite w' /\ SameAddress w w' /\ SameCore w w' /\
      ProgramOrder w w' /\ ProgramOrder w' i).

DefineMacro "STBEmpty":
  forall microop "w", (
    (IsAnyWrite w /\ SameCore w i /\ SameAddress w i /\ ProgramOrder w i) =>
    EdgeExists ((w, (0, WriteToMemHierarchy)), (i, MemoryStage), "STBEmpty")).

DefineMacro "ReadsFromMemory":
  % Read from "w", and there must not exist any writes w' to the same address
  % between w and i
  exists microop "w", (
    IsAnyWrite w /\ SameAddress w i /\ SameData w i /\
    EdgeExists ((w, (0, WriteToMemHierarchy)), (i, MemoryStage), "rf") /\
    ~(exists microop "w'",
      SameAddress i w' /\
      EdgesExist [((w , (0, WriteToMemHierarchy)), (w', (0, WriteToMemHierarchy)));
                  ((w', (0, WriteToMemHierarchy)), (i ,     MemoryStage         ))])).

DefineMacro "BeforeOrAfterEveryWriteToSameAddr":
  % Either before or after every write to the same physical address
  forall microop "w", (
    (IsAnyWrite w /\ SameAddress w i) =>
    (EdgeExists ((w, (0, WriteToMemHierarchy)), (i,     MemoryStage         ), "ws*;rf") \/
     EdgeExists ((i,     MemoryStage         ), (w, (0, WriteToMemHierarchy)), "fr"))).
```

Figure 5.3: PipeCheck model of the microarchitecture of Figure 5.2a, 1/3.

```
Axiom "WriteSerialization":
forall microops "i1", forall microops "i2",
((~SameMicroop i1 i2) /\ IsAnyWrite i1 /\ IsAnyWrite i2 /\ SameAddress i1 i2) =>
(EdgeExists ((i1, (0, WriteToMemHierarchy)), (i2, (0, WriteToMemHierarchy)), "ws") \/
 EdgeExists ((i2, (0, WriteToMemHierarchy)), (i1, (0, WriteToMemHierarchy)), "ws")).

Axiom "Reads":
forall microops "i",
(OnCore c i /\ IsAnyRead i) =>
EdgesExist [((i, Fetch      ), (i, Decode     ), "path");
            ((i, Decode     ), (i, Execute    ), "path");
            ((i, Execute    ), (i, MemoryStage), "path");
            ((i, MemoryStage), (i, Writeback  ), "path")]
/\
(
  ExpandMacro STBFwd \/
  (
    ExpandMacro STBEmpty /\
    (
      ExpandMacro ReadsFromMemory /\
      ExpandMacro BeforeOrAfterEveryWriteToSameAddr
    )
  )
).

Axiom "Writes":
forall microops "i",
OnCore c i =>
IsAnyWrite i =>
EdgesExist [((i, Fetch           ), (i,     Decode              ), "path");
            ((i, Decode          ), (i,     Execute             ), "path");
            ((i, Execute         ), (i,     MemoryStage         ), "path");
            ((i, MemoryStage     ), (i,     Writeback           ), "path");
            ((i, Writeback       ), (i,     LeaveStoreBuffer    ), "path");
            ((i, LeaveStoreBuffer), (i, (0, WriteToMemHierarchy)), "path")]
/\ (
% Only one at a time allowed out from the store buffer
forall microop "w", (
  (IsAnyWrite w /\ SameCore w i /\ ProgramOrder i w) =>
    EdgesExist [((i, (0, WriteToMemHierarchy)), (w, LeaveStoreBuffer), "STBOne")])
).

Axiom "mfence":
forall microops "f",
OnCore c f =>
IsAnyFence f =>
EdgesExist [((f, Fetch      ), (f, Decode     ), "path");
            ((f, Decode     ), (f, Execute    ), "path");
            ((f, Execute    ), (f, MemoryStage), "path");
            ((f, MemoryStage), (f, Writeback  ), "path")]
/\ (
  forall microops "w",
  ((IsAnyWrite w /\ SameCore w f /\ ProgramOrder w f) =>
     EdgeExists ((w, (0, WriteToMemHierarchy)), (f, Execute), "mfence"))).
```

Figure 5.4: PipeCheck model of the microarchitecture of Figure 5.2a, 2/3.

```
Axiom "RMW":
forall microop "w",
IsAnyWrite w => AccessType RMW w =>
(forall microops "i2", ProgramOrder w i2 => IsAnyRead i2 /\
  EdgeExists ((w, (0, WriteToMemHierarchy)), (i2, MemoryStage), "rmw")) /\
(exists microop "r",
  ConsecutiveMicroops r w /\ IsAnyRead r /\ AccessType RMW r /\
  ~exists microop "w'",
    IsAnyWrite w' /\ SameAddress w w' /\
    EdgesExist [((r,      MemoryStage         ), (w', (0, WriteToMemHierarchy)));
                ((w', (0, WriteToMemHierarchy)), (w , (0, WriteToMemHierarchy)))]).

Axiom "PO/Fetch":
forall microops "i1",
forall microops "i2",
(OnCore c i1 /\ OnCore c i2 /\ ProgramOrder i1 i2) =>
EdgeExists ((i1, Fetch), (i2, Fetch), "po").

Axiom "Decode_stage_is_in-order":
forall microops "i1",
forall microops "i2",
EdgeExists ((i1, Fetch),  (i2, Fetch)) =>
NodesExist [(i1, Decode); (i2, Decode)] =>
EdgeExists ((i1, Decode), (i2, Decode), "ppo_uarch").

Axiom "Execute_stage_is_in-order":
forall microops "i1",
forall microops "i2",
EdgeExists ((i1, Decode),  (i2, Decode)) =>
NodesExist [(i1, Execute); (i2, Execute)] =>
EdgeExists ((i1, Execute), (i2, Execute), "ppo_uarch").

Axiom "Memory_stage_is_in-order":
forall microops "i1",
forall microops "i2",
EdgeExists ((i1, Execute),    (i2, Execute)) =>
NodesExist [(i1, MemoryStage); (i2, MemoryStage)] =>
EdgeExists ((i1, MemoryStage), (i2, MemoryStage), "ppo_uarch").

Axiom "Writeback_stage_is_in-order":
forall microops "i1",
forall microops "i2",
EdgeExists ((i1, MemoryStage), (i2, MemoryStage)) =>
NodesExist [(i1, Writeback);  (i2, Writeback)] =>
EdgeExists ((i1, Writeback), (i2, Writeback), "ppo_uarch").

Axiom "STB_FIFO":
forall microops "i1",
forall microops "i2",
EdgeExists ((i1, Writeback),   (i2, Writeback)) =>
NodesExist [(i1, LeaveStoreBuffer); (i2, LeaveStoreBuffer)] =>
EdgeExists ((i1, LeaveStoreBuffer), (i2, LeaveStoreBuffer), "ppo_uarch").
```

Figure 5.5: PipeCheck model of the microarchitecture of Figure 5.2a, 3/3.

an axiom asserting that pairs of x86-TSO microops corresponding to an x86-TSO macroop with a LOCK prefix asserted (i.e., atomic read-modify-write instructions) are indeed atomic. Second, it asserts that two instructions related by `ProgramOrder` pass through the fetch stage in that order. Lastly, it provides a set of local reordering guarantees for each remaining stage in the pipeline, in this case representing the fact that this pipeline is in-order.

Some of the edges specified in Figure 5.3 are directly analogous to the reads-from (`rf`), from-reads (`fr`), and write serialization (`ws`) edges defined in Section 5.2. Each is interpreted in terms of the performing location for each microop with respect to the destination core. Some may even have multiple legal interpretations; for example, `rf` may be interpreted in the sense of forwarding from the store buffer or in the sense of reading from memory. In other cases, an architecture edge may not have an analogous edge. Note how the architecture-level `ppo` is omitted entirely in favor of a series of finer-grained `ppo_uarch` edges. Lastly, there may be microarchitecture-level edges which have no analogous architecture-level edges. For example, the "`path`" and "`STBEmpty`" edges do not directly correspond to any of the edge types described in Figure 5.1b.

While the complexity of a model depends on the complexity of the underlying microarchitecture, the complexity of the model does not necessarily scale linearly with the complexity of the microarchitecture. The model of Figures 5.3 to 5.5 is a relatively simple pedagogical example. Nevertheless, complex real-world models may be analyzed in a very similar manner; our PipeCheck model of the OpenSPARC T2 industry-strength processor is only just under twice as long as the model of our simple pedagogical pipeline. Sections 5.6 and 6.6.4 demonstrate how well the PipeCheck approach scales, both in terms of the ability to describe complex real-world microarchitectures and the ability to verify their correctness very quickly.

Furthermore, in keeping with standard tradeoffs in the field of model checking, the contents and granularity of the model are somewhat subject to the discretion of the user. Finer-grained models (i.e., ones with nodes modeling comparatively more distinct locations or events) will generally be closer to the behaviors of the register transfer level (RTL) definition of the microarchitecture. This will in turn generally mean that verification of the validity of each axiom will be comparatively easier. However, the graphs (and graph counts) will also likely be larger, and so verification time will increase. On the other hand, coarser-grained models (with fewer distinct locations/events) are likely faster to analyze, but the user then takes on a larger burden in ensuring that the coarser axioms are still truly representative of the underlying microarchitecture.

5.3.3 Transitivity of μhb Edges

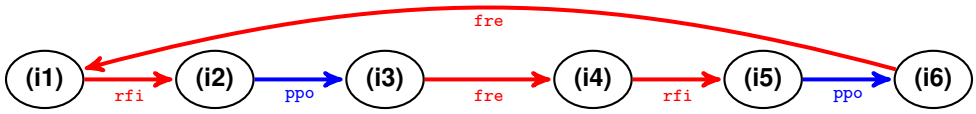
Section 2.2.4 demonstrated that in many memory models, transitivity of inter-instruction orderings is not always guaranteed. In other words, since there are in general many competing points of view by which orderings can be defined, it is possible that for instructions (i1), (i2), and (i3), (i1) may appear to happen before (i2), (i2) may appear to happen before (i3), and (i3) may appear to happen before (i1), all simultaneously. Likewise, Figure 2.8 demonstrated that edges in single-event axiomatic models cannot in general be transitively composed, because single-event models search for cycles only among edges with certain carefully-chosen labels, and the transitive closure of two edges with different labels may not itself always have its own well-defined label.

A major benefit of PipeCheck (and of many multi-event axiomatic models [MHMS+12]) is that all edges are in fact treated equally, regardless of their label or their reason for being added to the graph. As described in Section 2.2.2, in most single-event axiomatic models, the axioms search for cycles only among

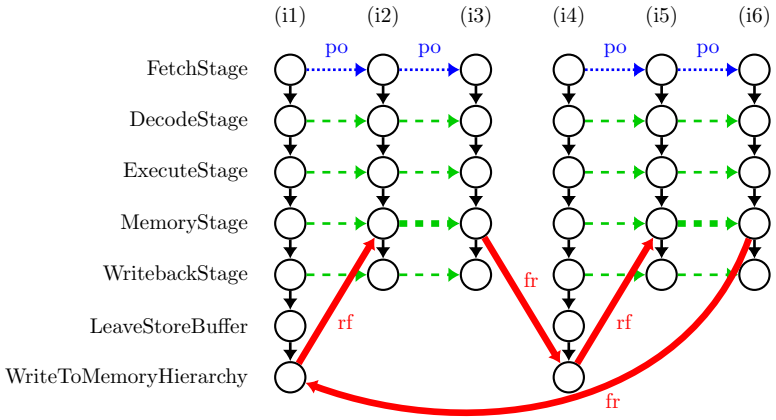
Thread 0		Thread 1	
(i1)	[x]←1	(i4)	[y]←1
(i2)	r1←[x]	(i5)	r3←[y]
(i3)	r2←[y]	(i6)	r4←[x]

Outcome: 0:r1=1, 0:r2=0, 1:r3=1, 1:r4=0
Permitted under TSO

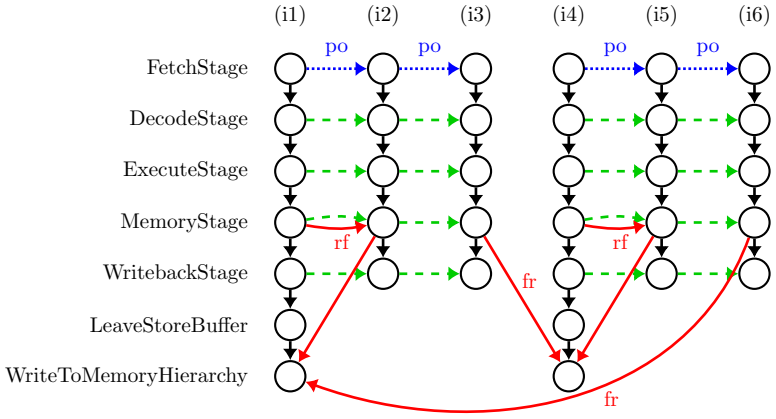
(a) Litmus test `iwp2.4/amd9`



(b) Architecture-level analysis. Even though there is a cycle, there is no cycle involving the subsets in Figure 5.1b, and so the outcome is permitted.



(c) Cyclic (and hence unobservable) μhb graph for an execution in which all reads skip the store buffer and access memory



(d) Acyclic (and hence observable) μhb graph for an execution in which reads (i2) and (i5) forward data from writes still in the store buffer

Figure 5.6: Analyzing litmus test `iwp2.4/amd9`.

edges with certain specific labels, and therefore edges with different labels must be treated differently. In PipeCheck, on the other hand, cycle checking is performed using all edges in the graph regardless of their labeling. As a consequence, it is always legal to take the transitive closure of μhb edges[3]. Microarchitecture-level transitivity is respected because each μhb edge represents either a local ordering *at a particular microarchitectural location* or a communicated message (i.e., a Lamport clock [Lam78] "happens before").

Note that microarchitecture-level transitivity does *not* imply architecture-level transitivity, and that the presence of architecture-level cycles does not imply the presence of microarchitecture-level cycles. Similarly, causality (Section 2.2.4) is also not a consequence of μhb transitivity. This lack of implications above ensures that PipeCheck can describe architecture-level memory models which are not themselves transitive or causal (e.g., Power [IBM13]).

For example, Figure 5.6a shows a litmus test which demonstrates that the presence of store buffering can be observed by software. This litmus test was first analyzed in Section 2.2.1. Figure 5.6b shows that naive architecture-level analysis would produce a cyclic graph. However, neither of the two specially-chosen hb graph subsets used to define TSO in Figure 5.1b are cyclic, and hence the outcome remains permitted. Figures 5.6c and 5.6d show two μhb graphs (i.e., two of the various legal graphs) for the same litmus test. Figure 5.6c shows that a scenario in which all inter-instruction communication takes place through memory results in a cyclic graph. However, Figure 5.6d shows that the scenario in which two of the loads read from the store buffer is in fact acyclic and hence observable. This last case shows an example of how the presence of an architecture-level cycle (Figure 5.6b) does not imply the presence of a microarchitecture-level cycle (Figure 5.6d).

[3]For legibility, we do not draw the full transitive closure in our figures. Nevertheless, edges resulting from the transitive closure of any other edges in the graph can always be inferred.

This analysis highlights two facts. First, μhb graphs restore an intuitive one-to-one correspondence between presence/absence of a cycle and non-observable/observable outcomes, respectively. Although it may not matter to automated analysis algorithms, this one-to-one correspondence makes it easier for humans to reason about μhb graphs being drawn. Second, μhb graphs directly depict the *manner* in which a particular outcome may be observed. In other words, it is clear from Figure 5.6d that the execution is observable only when loads are allowed to forward data from the store buffer. Although μhb graphs have significantly more nodes and edges than hb graphs, the association of a particular physical location with each μhb node means that each edge is described by the source and destination nodes alone.

5.3.4 Modeling Microarchitectural Optimizations

High-performance processors implement a number of microarchitecture-level optimizations which are functionally invisible to the user (and to architecture-level memory models) but which are nevertheless very important to correct memory ordering behavior in practice. Many such optimizations map very naturally into μhb graphs and PipeCheck axioms. Others require somewhat more abstraction. We discuss some particular cases below.

One potential point of difficulty comes about in specifying models for superscalar pipelines and/or components such as caches with multiple ports. These can be most directly modeled in PipeCheck by simply treating each component with multiple lanes as distinct locations and by adding axioms to enforce any enforced cross-lane orderings. However, using this strategy, the state space would grow quickly, as the microarchitecture specification would need to explicitly model both/all possibilities, leading to an exponential number of possibilities if analysis is done naively. Alternatively, one could assign an arbitrary priority between lanes sharing a location to resolve ties, or one could even imagine μhb edges which indicate "happens before or

at the same time as". However, these deviate further from the basic approach of PipeCheck, and so we do not consider them further.

Many microarchitectures use various forms of speculation to improve performance. PipeCheck models this simply by not including squashed speculated events in μhb graphs, as the consistency model imposes no requirements on squashed instructions. If a squashed instruction is later replayed, then that replay will be included in the graph (unless it is itself squashed). If a squashed instruction is not replayed (e.g., because it was only executed due to an incorrect speculation), then it simply does not appear.

On the other hand, PipeCheck models can and do explicitly model correctly-speculated behaviors, as the implementation of the speculation mechanism(s) does in general affect the values returned by loads and hence is relevant to consistency verification. For example, if a load returns the value written by a speculative store, or if two load may speculatively execute out of order, then the PipeCheck model of the microarchitecture in question must 1) capture the speculation explicitly, and 2) ensure that the loads only commit if their returned values are legal according to the memory model. This enforcement can take place in different ways. For example, squash events may themselves imply μhb behavior, in the sense that the replayed instruction will be guaranteed to restart after the event that caused the squash to take place. Section 5.7 describes various examples in more detail. Out-of-order execution and correct speculation both map very naturally into PipeCheck. To enable them, the axioms of a model must simply be relaxed to take into account the fact that some events might happen earlier than they might otherwise occur under in-order, non-speculative circumstances. Of course, relaxing the microarchitecture too much will result in violations of the architecture-level requirements. Often, then, the weakening of some axioms will require the addition of axiom(s) representing some "fallback" mechanism which watches for illegal reorderings and handles them in some

way. Section 5.7.1 provides a detailed example of how speculative load reordering is modeled in PipeCheck.

5.4 Verification Flow

While the previous section described how PipeCheck models are specified, this section describes how the correctness of each model is verified.

5.4.1 Litmus Test-Based Verification

Litmus tests are a standard and widely-used tool in verification of consistency models [AMT14, HVML04, MHMS+12, OSS09a, SSA+11]. As described in Section 2.1.4, litmus tests are (usually very short) programs designed to test particular rules or subcases for a consistency model. Due to the inherent complexity of defining even simple consistency models and/or due to incomplete or even incorrect documentation, even programs as short as five or six instructions (e.g., `amd6/iriw` [BA08], `n4/n5/n6` [OSS09a], A-cumulativity tests [AFI+09], `coRSDWI/mp+dmb+fri-rfi-ctrlisb` [AMT14]) can be very difficult to analyze properly. Nevertheless, the ability to execute litmus tests on real hardware allows them to serve as a valuable means for checking that a model is sound with respect to an actual implementation, and/or vice-versa.

At a high-level, verification proceeds according to the flow diagram of Figure 5.7. The PipeCheck tool itself takes in two inputs. First, it takes a microarchitecture model specification. These specifications are written in the domain-specific language of Section 5.3.2. Second, PipeCheck takes in a litmus test written using a pre-existing and standardized syntax [AMSS11]. Given these inputs, the automated constraint solver of Section 5.4.2 is used to determine whether the specification admits an acyclic

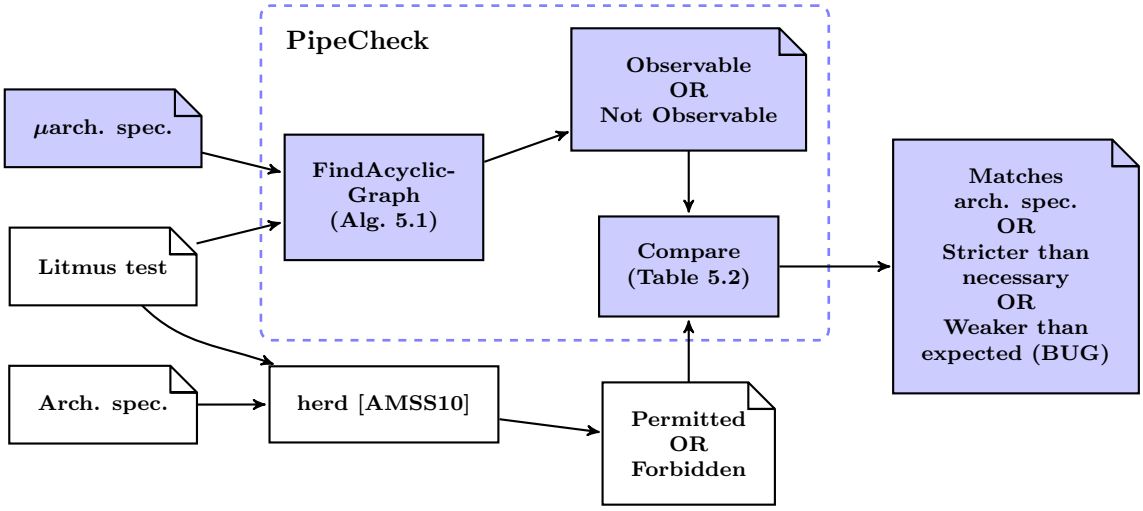


Figure 5.7: PipeCheck block diagram and toolflow. Pre-existing components are shown in white; shaded components were created for PipeCheck.

μhb graph for the litmus test. If so, the outcome is *observable* on the given microarchitecture model. If not, the outcome is *not observable*.

The next step depends on whether there exists an architecture-level specification against which the above outcome can be compared. Ideally, we would compare the above result to a specification of the expected architecture-level behavior to ensure that no forbidden outcomes are observable. However, an unfortunate reality of modern hardware development is that formal (or even informal) architecture-level MCM specifications simply may not exist. Accelerators in general, and GPUs in particular, serve as currently-prominent examples: GPU hardware specifications from industry are often weak and informal-to-nonexistent [AMD12, HSA13, NVI13b, NVI15]. Academic research has demonstrated that these specifications are both fundamentally flawed and often implemented incorrectly, meaning that they are clearly unsuitable as an absolute statement of correctness [ABD$^+$15]. Likewise, modern GPU toolchains differ from CPU toolchains in that the GPU flow uses just-in-time (JIT) compilation from an software intermediate representation to a microarchitecture-specific instruction set, meaning that the traditional notion of an architecture-level abstraction layer

	Microarchitecturally	
Architecturally	Observable	Not Observable
Permitted	OK	OK (but stricter than necessary)
Forbidden	Bug in model and/or microarchitecture	OK

Table 5.2: Approach to verifying a microarchitecture-level model against an architecture-level specification

does not even exist [HSA13, NVI13b]. Because of this, we consider two slightly distinct workflows, each of which is described in turn below.

When an Architecture-Level MCM Specification is Available. When possible, PipeCheck compares the expected behavior (according to the architecture-level model) to the observed behavior (according to the microarchitecture-level model). To determine the expected behavior, PipeCheck currently uses the `herd` tool, a pre-existing architecture-level analysis tool [AMT14], to determine whether the outcome proposed by the litmus test is permitted or forbidden on that architecture. The `herd` tool takes in the same litmus test and an architecture-level memory model specification written in the `cats` language, which has itself been used to describe a wide variety of memory models. However, PipeCheck and `herd` are decoupled, so other tools or models could easily be used as well.

The set of possible analysis outcomes is summarized in Table 5.2. The two most common outcomes are those in which the expected and observed behaviors correspond. Outcomes which are architecturally permitted and microarchitecturally observable are correct, as are outcomes which are architecturally forbidden and microarchitecturally unobservable.

An architecturally-permitted but microarchitecturally-unobservable outcome is not in itself an incorrect result. Such a situation may simply mean that the pipeline is stronger than strictly necessary. It may imply a performance penalty, given that stricter consistency models are correlated with lower performance, but the situation

generally is legal. It may also imply that one of the axioms in the model is stricter than what the underlying microarchitecture actually permits, which in turn implies that the overly-strong constraint should be removed from the model. In any case, care should be taken to ensure that the microarchitecture ensures at least some outcomes (e.g., at least sequentially consistent ones), as it is possible to (usually unintentionally) write a model which simply forbids all outcomes. A microarchitecture model which forbids all outcomes should not be considered correct.

An architecturally-forbidden but microarchitecturally-observable outcome does indicate a problem. The problem may be that the model is underconstrained: that some ordering enforced by the microarchitecture is not currently reflected in the model. In other cases, if the model is faithful, then the unexpected outcome may actually correspond to a consistency bug in the microarchitecture. In either case, in contrast to some constraint solvers, PipeCheck returns the explicit μhb graph corresponding to the unexpected outcome. An architect can then study this graph to determine where the fault lies and how best to correct it.

When an Architecture-Level MCM Specification is Not Available. Although a specification of expected behavior may not available, it is still possible to simply analyze the behavior of a microarchitecture independently of any correctness specification. A microarchitecture model, once properly refined, may even serve as the inspiration for a later architecture-level specification. In this sense, our hope is that PipeCheck is used both as an exploration tool and as a verification tool.

It is also possible to use PipeCheck to reason about correctness with respect to a level of abstraction even higher than the architecture level. For example, one might reason about the correctness of a particular compiler/microarchitecture combination with respect to a given software-level specification. Previous work has proven the correctness of mappings and/or compilers in the past [BMO$^+$12, BSDA14, Ler09, PVJ15]. However, in such cases, the framework must explicitly address the way in

Algorithm 5.1 PipeCheck Constraint Solver

Given: μarch spec f_{FOL}, as defined using the DSL of Section 5.3.2
Given: litmus test t

// Eliminate quantifiers by explicitly enumerating over the (finite) domain(s) in t
$f_{\text{prop}} \leftarrow$ EliminateQuantifiers(f_{FOL}, t)

// Convert f_{prop} into the form used by the solver
$f_{\text{prop,nnf}} \leftarrow$ ConvertToNegationNormalForm(f_{prop})

// Algorithm 5.2, starting with an empty graph
return FindAcyclicGraph($\varnothing$, $f_{\text{prop,nnf}}$)

which higher-level constructs are mapped or compiled onto microarchitecture-level primitives. As such, the consideration of such mappings is outside of the scope of this thesis.

5.4.2 Automated Constraint Solver Algorithm

The PipeCheck constraint solver is inspired by the Davis-Putnam-Logemann-Loveland (DPLL) algorithm that forms the core of many existing boolean satisfiability (SAT) solvers [DLL62, MMZ+01]. Pseudocode for the solver is given in Algorithms 5.1 and 5.2. As Figure 5.7 and Algorithm 5.1 show, the solver takes in two inputs: the microarchitecture specification (as specified using the DSL of Section 5.3.2), and a litmus test specified using the pre-existing `litmus` format [AMS+12]. The solver is divided into two phases: a preprocessing step and the core solver. We address each below in turn.

Solver Algorithm: Preprocessing Phase. The goal of the preprocessing phase is to parse the two inputs (the μarch spec and the litmus test) into a form that is suitable for automated satisfiability checking. The preprocessing phase is divided into the two steps discussed below.

First, because the formula is being applied to a particular litmus test, the domains over which variables in the formula may be quantified—microops and threads—are

concrete and finite. Therefore, to ease the job of the constraint solver, the quantifiers are eliminated: each `forall` quantifier becomes a conjunction over its domain, and each `exists` quantifier becomes a disjunction over its domain. Likewise, all predicates except `NodesExist` and `EdgesExist` are evaluated to either true or false. The `NodesExist` and `EdgesExist` predicates become simple propositions. The above process produces a propositional formula representing the constraints of the microarchitecture model as applied the litmus test in question.

Second, the propositional formula generated above is converted into negation normal form (NNF), a form in which the NOT operator may only be applied to individual propositions (as opposed to clauses) and in which only AND and OR operators are allowed as connectives. As a design choice, the formula is not converted into conjunctive normal form (CNF) as would be required by standard SAT solvers. Our implementation (Section 5.4.3) provides the ability at any step to view the partially-completed graph and the tree of remaining constraints. As such, we find that maintaining the structure in the form originally specified by the user is a significant help during the debugging process.

Once the above two steps are completed, the NNF formula is passed (along with the empty graph) as the arguments to Algorithm 5.2 below.

Solver Algorithm: FindAcyclicGraph. At a high level, the algorithm follows a recursive backtracking approach. Candidate solutions (i.e., acyclic graphs) are built up incrementally by adding new edges to the graph. At various points in the search process, there may be two (or more) possible choices to take, and the solver will consider each choice one by one. If at any point the graph under construction becomes cyclic and/or the set of remaining constraints becomes unsatisfiable, all remaining solutions derived from that point are discarded, and the algorithm backtracks to a previous point to explore a different part of the solution space.

Algorithm 5.2 FindAcyclicGraph

```
Given: μhb graph g
Given: formula f of remaining constraints, in negation normal form

repeat
  // Prune terms in f that are redundant/invalid given g
  f ← PropagateConstraints(f, g)

  // Find guaranteed literals, i.e., nodes or edges guaranteed to be in every possible
  // solution derived from the current state
  l ← GuaranteedLiterals(f)
  g ← g ∪ l

  // If g has become cyclic, prune the rest of this subtree
  if Cyclic(g) then
    return (NoSolutionFound, ∅)

  // If new guaranteed literals were found, iterate to keep pruning the search space
until l = ∅

if AllConstraintsSatisfied(f) then
  // The iteration has converged, and g is acyclic (because otherwise it would have
  // been pruned above). Hence we have found a valid acyclic graph.
  return (Observable, g)
else if RemainingConstraintsUnsatisfiable(f) then
  // Backtrack and try a different candidate
  return (NoSolutionFound, ∅)
else
  // Find a list of "branching edges"
  for all b ∈ FindBranchingEdges(f) do
    // Recurse using each branching edge candidate
    (result, g′) = FindAcyclicGraph(g ∪ b, f)

    // If a valid scenario was found during recursion, stop searching and return it
    if result = Observable then
      return (Observable, g′)
    else
      // Otherwise, backtrack to the current state and try another candidate
      continue

  // If none of the candidates returned a valid acyclic graph, then give up on this
  // subtree and backtrack one step further
  return (NoSolutionFound, ∅)
```

Each iteration is composed of two steps: an inner simplification loop and a recursive subcase exploration step. The simplification step relies on the concept of a *guaranteed literal*—a literal (a proposition or its negation) which must be true given its position within the formula. Guaranteed literals are analogous to unit clauses, or clauses containing only one literal, in CNF formulas: since every clause in a CNF formula must hold true, a literal in a CNF unit clause must itself hold true. The value of guaranteed literals (and of unit clauses) is that they allow for *unit propagation*, a rule for simplifying a propositional formula by simplifying other occurrences of that literal in the formula. Under unit propagation, every non-negated occurrence of the same literal is can be replaced with boolean true, and every negated occurrence of the same literal is unsatisfiable and can be replaced by boolean false. In PipeCheck, multiple instantiations of the same edge or node can likewise be replaced by boolean true, even when the edges may have different labels, as discussed in Section 5.3.3.

The μhb graph-centric focus of PipeCheck introduces two key distinctions from standard unit propagation/boolean constraint propagation. First, because PipeCheck formulas are not in CNF, literals in NNF unit clauses are not inherently guaranteed to be true, as they may be part of a broader disjunction. Nevertheless, this makes no difference in practice, as in either case literals are considered to be guaranteed if and only if they are reachable only through conjunctions. Second, the assignment of a truth value to a given literal (i.e., edge or node) may have a direct impact on the satisfiability of other literals. For example, the addition of an edge to the graph may cause other literals to become unsatisfiable, as the addition of the second edge may (in conjunction with the first edge) complete a cycle. Edge literals and node literals may affect each other in similar ways. In this sense, PipeCheck behaves as a primitive SMT solver for the theory of directed acyclic graphs, rather than as a pure SAT solver [BSST09].

As noted by existing SAT solvers, constraint propagation often has a large and cascading effect, and in practice most of the runtime of the solver may be spent in the constraint propagation phase [MMZ+01]. PipeCheck models tend to be no different in this regard; most models tend to have many guaranteed nodes and edges. For example, each instruction in a program will always have a fetch to decode edge, decode to execute edge, and so on, and so these edges will be marked as guaranteed from the beginning. Furthermore, the sequence of per-pipeline stage preserved program order axioms found in many pipeline definitions (e.g., as in Figure 5.5) will become guaranteed one iteration at a time. Often, in PipeCheck, this process runs to completion even without requiring any of the recursive calls described below.

Once the constraint propagation process has converged, there are three possibilities. First, if the remaining constraints reduce to boolean true, then a valid acyclic graph has been found, and the algorithm completes. Second, if the remaining constraints reduce to boolean false, then no solution derived from this point is valid, so the algorithm discards the current state and backtracks to the previous step. Third, if the remaining constraints reduce neither to boolean true nor to boolean false, then the constraints represent the set of possible solutions derived from the current state. In the latter case, the algorithm recurses to search for a solution among the set of remaining possibilities.

The recursion into subcases proceeds as follows. First, the algorithm searches among the remaining constraints for a CNF clause: a set of unassigned literals whose disjunction holds true according to the remaining constraints, as a clause in a CNF formula would. In other words, a CNF clause is a set of edges such that at least one of the edges in the set must be present in all solutions derived from the current point. For each candidate literal in the CNF clause, the algorithm recurses, passing the remaining constraints and the graph with the candidate edge(s) added. Each candidate is considered in turn, until a solution is found (at which point there is no

need to continue searching) or until all possibilities have been exhausted (in which case the algorithm backtracks further). If the algorithm finds a solution at any point during its search, it returns Observable, along with the graph that satisfies the constraints. If the algorithm backtracks all the way to the beginning without finding a solution, then the constraints are unsatisfiable for the given litmus test, and the algorithm returns NotObservable.

5.4.3 Software Implementation

To aid in exploration and verification of microarchitecture-level memory models, we implemented the PipeCheck flow of Figure 5.7 as an automated toolchain. We use a mixture of Coq, OCaml, and external tools. The core analysis routines (constraint checking, cycle detection, iteration until convergence, etc.) are written in Coq, an interactive theorem prover [The04]. The use of Coq allows propositions about correctness to be formally stated and proven, and it provides an extremely strong guarantee of reliability. It also allows our framework to be integrated with existing open-source architecture-level frameworks also written in Coq [AMSS10, Alg12]. At this time, however, as a result, we have not completed any formal proofs; see Section 5.4.6.

The interface to PipeCheck is written in OCaml. The core analysis routines are automatically extracted from their purely-functional Coq specifications into a set of OCaml functions using built-in features of the Coq framework. We then write an external interface to these analysis routines using native OCaml. This includes parsers for litmus tests and microarchitecture models (such as the ones presented in Figures 5.3 through 5.5). The entire framework is then compiled into a standalone tool, resulting in the flow depicted in Figure 5.7.

5.4.4 Runtime

We chose to write the analysis routines in Coq to allow the analysis to be formally specified and analyzed. However, this does come with a cost in terms of performance. Coq routines and data structures are generally optimized to be amenable to formal analysis rather to be high-performance, and so the runtimes of many operations are both absolutely and asymptotically slower than might be possible when coding directly in OCaml (or any other language). Nevertheless, even with the overheads of naive Coq functions and data structures and of naive constraint solving, the graph counts and graph sizes remain small enough to be tractable, and so the analysis remains very practical. We provide quantitative evidence of this in Section 5.6. As a result, we consider the current balance between formalism and practical usability to be an important feature of PipeCheck.

The number of graphs each algorithm enumerates varies with the number of instructions in the test and the size of each pipeline. While potentially exponential in the worst case, the absolute numbers for each quantity are very small: long pipelines contain just dozens of stages, and the longest litmus test of the suite we survey contains eight instructions. Furthermore, the algorithm can (and does) terminate early if it finds an observable outcome. Finally, the graph checking can be trivially parallelized. This ensures that PipeCheck remains fast in practice.

5.4.5 Caveats

As with any formal verification process, the verification outcome is only legal to the extent that the model faithfully represents the underlying microarchitecture. In particular, a forbidden but observed outcome indicates either a pipeline bug or an incomplete specification. If the model is missing an axiom representing some ordering that is in fact enforced by the hardware, then the axiom can be added and the verification process can be retried. On the other hand, if all behaviors actually enforced by

the pipeline are represented by some axioms in the model, then a verification failure does in fact represent an ordering bug in the underlying microarchitecture. In this case, the μhb edge which is determined to be missing can be enforced by modifying the microarchitecture accordingly; see Section 5.7.2 for an example.

5.4.6 Future Work: Formal Equivalence/Implication

One drawback of the litmus test-based approach in general is that it can suffer from a lack of coverage. Although test suites are large, they are not guaranteed to test every possible scenario, and so it is possible that some bug in a microarchitecture could go undetected due to a lack of coverage. This situation in fact occurred more than once during the development of PipeCheck; we have continued to add new microarchitecturally-inspired litmus tests throughout the development process. Researchers have studied the cost of enumerating a fully-comprehensive suite of litmus tests for a given architecture. However, even after careful elimination of redundancies that lead to two-orders-of-magnitude reductions in test counts, the numbers still grow into the tens of thousands and beyond [MHAM10, MHAM11].

A compelling alternative to using litmus tests would be to formally prove that a microarchitecture is sound with respect to the given architectural memory model. The theorem of correctness would state that any program outcome observable on the microarchitecture is allowed by the architecture specification:

$$\forall(\mathrm{p} \in \mathrm{programs}),$$
$$\exists \mathrm{g}_{\mu\mathrm{hb}}, \mathrm{IsValidScenario}_{\mu\mathrm{arch}}(\mathrm{p}, \mathrm{g}_{\mu\mathrm{hb}}) \implies$$
$$\exists \mathrm{e}, \mathrm{IsValidExecution}_{\mathrm{arch}}(\mathrm{p}, \mathrm{e})$$

Or, contrapositively, any outcome forbidden by the architecture should be forbidden by the microarchitecture as well. The caveat of Table 5.2 would still hold:

Pipeline	Lines of Code	Notes
RISC 5-Stage (w/o SB)	136	In-order pipe, unordered memory
RISC 5-Stage (Figs. 5.3-5.5)	167	In-order pipe, unordered memory
gem5 O3	190	Simulator, OoO pipe, unordered memory
OpenSPARC T2	233	Industry-strength, in-order pipe, mostly unordered memory

Table 5.3: Microarchitectures analyzed in this chapter.

permitted outcomes need not be observable on the microarchitecture. For example, to prove a microarchitecture correct with respect to the TSO memory model, one might use the formalization of Alglave et al. [Alg12, AMT14]:

$$\forall(\mathrm{p} \in \mathrm{programs}),$$
$$\exists \mathrm{g}_{\mu\mathrm{hb}}, \mathrm{IsValidScenario}_{\mu\mathrm{arch}}(\mathrm{p}, \mathrm{g}_{\mu\mathrm{hb}}) \implies$$
$$\exists \mathrm{rf}, \exists \mathrm{ws},$$
$$\mathrm{acyclic}(\mathrm{ppo}(\mathrm{p}) \cup \mathrm{mfence}(\mathrm{p}) \cup \mathrm{rf} \cup \mathrm{ws} \cup \mathrm{fr}(\mathrm{rf}, \mathrm{ws})) \wedge$$
$$\mathrm{acyclic}(\mathrm{poloc}(\mathrm{p}) \cup \mathrm{rf} \cup \mathrm{ws} \cup \mathrm{fr}(\mathrm{rf}, \mathrm{ws}))$$

For this thesis, we chose to focus on building a practical tool with broad applicability, and this goal contrasted to the often years-long process of properly formalizing hardware, microarchitecture, and mappings between them. We therefore leave full formalization and formal proofs to future work. See Section 7.1.2 for details.

5.5 Experimental Methodology

To demonstrate the effectiveness of PipeCheck, this chapter focuses on verification of processors implementing the TSO consistency model. TSO imposes non-trivial preserved program order requirements on all memory operations, making verification of TSO a particularly interesting target. Furthermore, its widespread use on x86

and other platforms make its verification very important. However, PipeCheck can also be used to model implementations of Power, ARM, or other architectures which have weaker PPO requirements and use fences or dependencies as synchronization primitives.

Table 5.3 summarizes the four microarchitectures we survey in our results. The first two are the five-stage RISC pipeline of Figure 5.2a both without and with a store buffer. The former is effectively a sequentially-consistent core, meaning that some of the litmus test outcomes permitted under TSO should not be observable. These two microarchitectures reflect the size of pipelines that might be used in classrooms or as small embedded cores. The third is the gem5 O3 simulated pipeline (v10013) [BBB+11]. This represents an average-sized core and demonstrates how simulated cores are also amenable to analysis. Finally, we describe the OpenSPARC T2 pipeline, representing a well-documented industry microarchitecture [Sun07].

We analyze a comprehensive set of TSO litmus tests from Intel and AMD manuals, academic studies, random test generators, and custom additions [AMD13, AMS+12, Int07, OSS09b]. Each litmus test was analyzed on a four core version of each pipeline, as none of our tests required more than four cores. We execute the extracted OCaml code and collect timing results using an Intel Xeon E5-2667 v3 server processor.

5.6 Results Across Litmus Tests

Table 5.4 shows the results of running the litmus test suite on each pipeline. Although we run the whole suite, only a core subset of the results are reproduced in the table; the rest of the suite behaves similarly. In this table, individual litmus tests are depicted as rows. For each row, the table shows whether TSO forbids or permits the outcome proposed by the test, and then shows its observability on the microarchitectures considered.

Litmus test	Expected	Observed				
	TSO	RISC (no SB)	RISC (w/ SB)	gem5 O3	gem5 (fixed)	Open-SPARC
iwp2.1/amd1/mp	**Forbid**	=	=	**O**[2]	=	=
iwp2.2/amd2/lb	**Forbid**	=	=	=	=	=
iwp2.3a/amd4/sb	Permit	**N**[1]	=	=	=	=
iwp2.3b	Permit	=	=	=	=	=
iwp2.4/amd9	Permit	**N**[1]	=	=	=	=
iwp2.5/amd8/wrc	**Forbid**	=	=	**O**[2]	=	=
iwp2.6	**Forbid**	=	=	=	=	=
amd3	Permit	**N**[1]	=	=	=	=
amd6/iriw	**Forbid**	=	=	**O**[2]	=	=
n1	Permit	**N**[1]	=	=	=	=
n2	**Forbid**	=	=	=	=	=
n4	**Forbid**	=	=	=	=	=
n5	**Forbid**	=	=	=	=	=
n6	Permit	=	=	=	=	=
n7	Permit	**N**[1]	=	=	=	=
rwc	Permit	**N**[1]	=	=	=	=

[1]Implementation more restrictive than TSO requires.
[2]Indicates the presence of a bug. See Section 5.7.2.

Table 5.4: Summary of litmus test results. "=": Matches expected TSO outcome. "O": Observable. "N": Not observable.

In almost all cases, the microarchitecturally-observable behaviors correspond with the architecturally-specified behaviors. However, there are exceptions. For the RISC pipeline without a store buffer, six of the proposed litmus test outcomes are permitted under TSO but forbidden under sequential consistency. They should therefore not be observable on the simplest pipeline, as it was designed to execute in a sequentially consistent manner. The six rows with "N" markers confirm the hypothesis.

On the other hand, three of the test results demonstrate the presence of a bug in the original specification of the gem5 O3 pipeline. This bug is discussed in detail as a case study in Section 5.7.2. For now, we simply observe that a problem is immediately apparent from the fact that three of the tests fail. As discussed in Section 5.4.1, the finding could indicate either that the model was under-constrained with respect to the underlying hardware or that the underlying hardware was itself buggy. As Section 5.7.2 demonstrates, the latter turned out to be the case. For comparison, we present results for the corrected version of the pipeline as well.

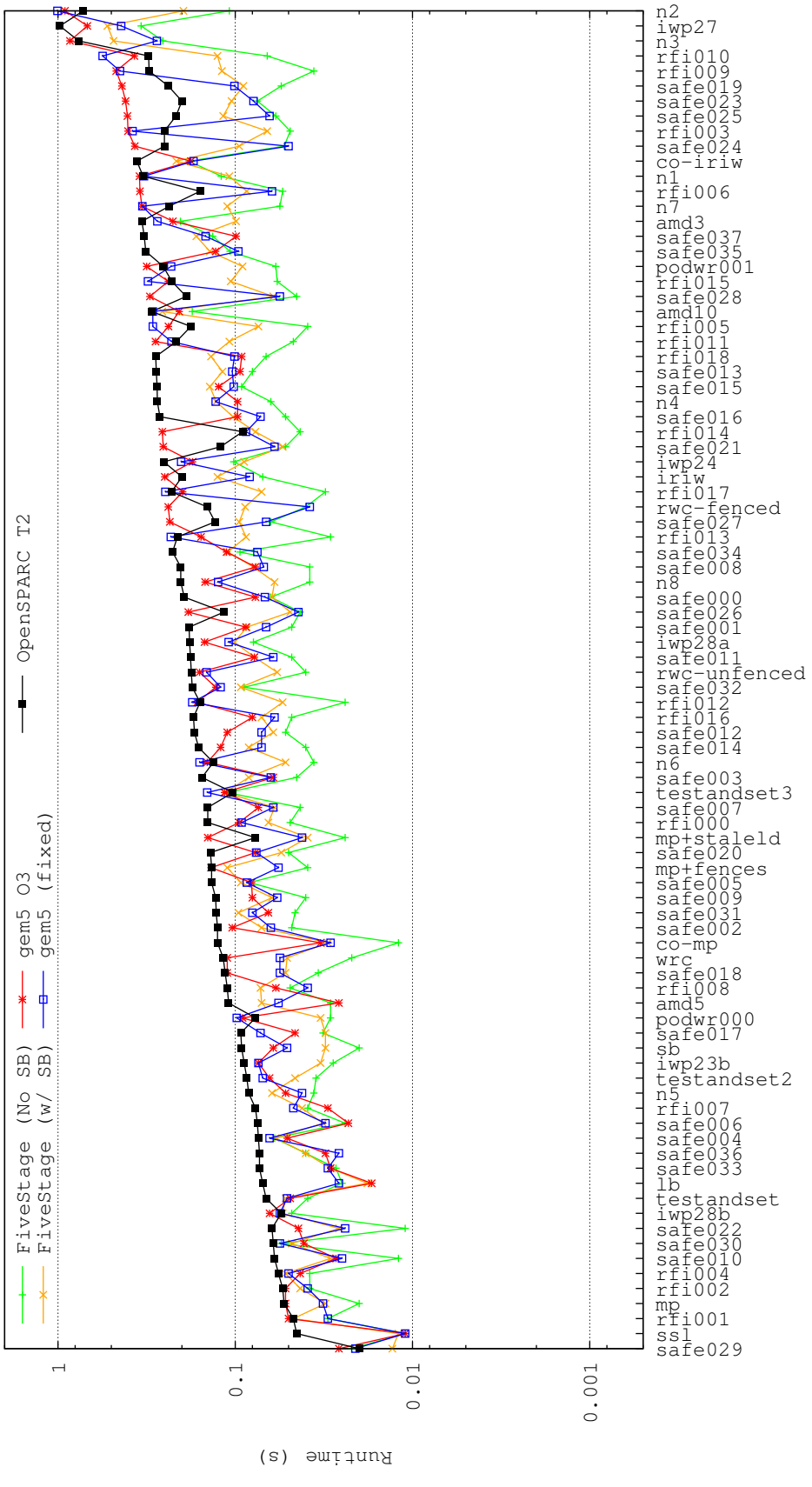


Figure 5.8: Verification Time Results (computed using extracted OCaml).

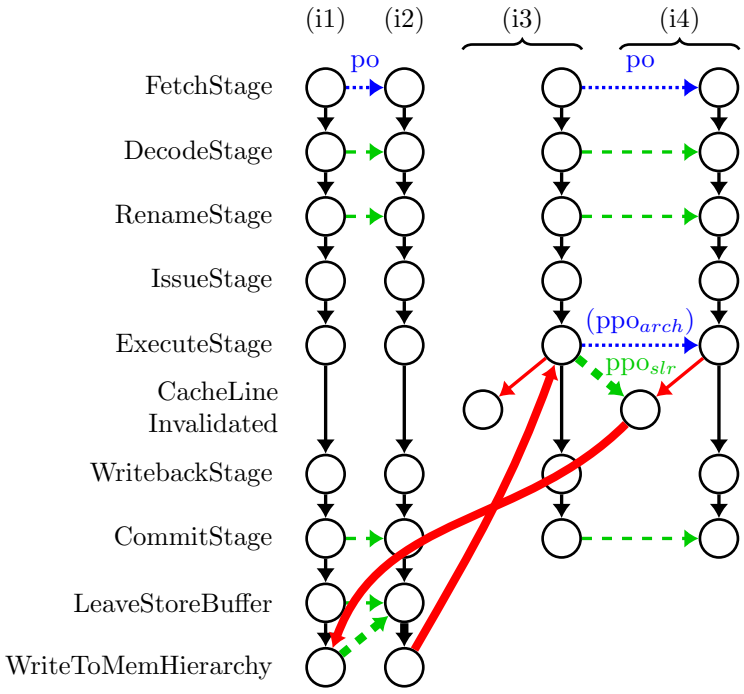
Figure 5.8 shows the time taken to complete the verification process for each pipeline. This figure includes the entire suite of litmus tests. Notably, the entire suite runs in less than a second for each pipeline. The fast runtimes allow PipeCheck to be used in an interactive manner, thereby greatly aiding in the debugging process. They also demonstrate that even though the code is written in Coq and optimized for verifiability rather than performance, the automated PipeCheck analysis remains very practical.

The performance of the PipeCheck solver progressed over time. Published early prototypes of the PipeCheck tool used naive, exhaustive, brute force exploration of the space. This remained scalable for early cases; runtimes were originally on the order of seconds to minutes [LPM14, LPM15]. However, as we scaled PipeCheck to run larger and/or more complicated cases (as in Chapter 6 and Section 7.1.1), we were motivated to improve the algorithm (and the generality) to improve the runtime. This resulted in a performance improvement of roughly two orders of magnitude.

Likewise, the generality of the microarchitecture specification format also improved greatly over time. The first version of PipeCheck was not flexible enough to handle atomic read-modify-write operations or fences of different varieties [LPM14, LPM15]. As a result, many of the tests shown in Figure 5.8 could not be run under the original model. Fortunately, the development of the PipeCheck DSL of Section 5.3.2 since the original publication allows for more flexible models, and we are now able to run 100% of the test suite.

5.7 Case Studies

Having defined the PipeCheck methodology, we now demonstrate its use by highlighting cases of particular interest.



(a) Speculative load reordering: although ppo$_{arch}$ is not enforced, a legal replacement ppo$_{slr}$ is enforced, and it completes the cycle.

5.7.1 Speculative Load Reordering

Many microarchitectures speculatively reorder load instructions for performance reasons [AMD13, GGH91, Int13a, SPA94b]. The key principle is that two loads l_1 and l_2 in program order can be speculatively reordered (i.e., l_2 can execute before l_1) as long as the value read speculatively by l_2 is the same as it would have been had l_2 in fact performed after l_1 (i.e., non-speculatively). One implementation, as used by the gem5 O3 pipeline [BBB+11] that we analyze in this thesis, is to hook into the cache coherence protocol. Namely, if a private cache line has not been overwritten or invalidated (due to cache replacement or an external invalidate request) since an earlier read of that line, then the core can safely assert that a subsequent read of that line would return the same value. On the other hand, if the cache line is invalidated, the core is conservative and assumes that the invalidate indicates a failed speculation.

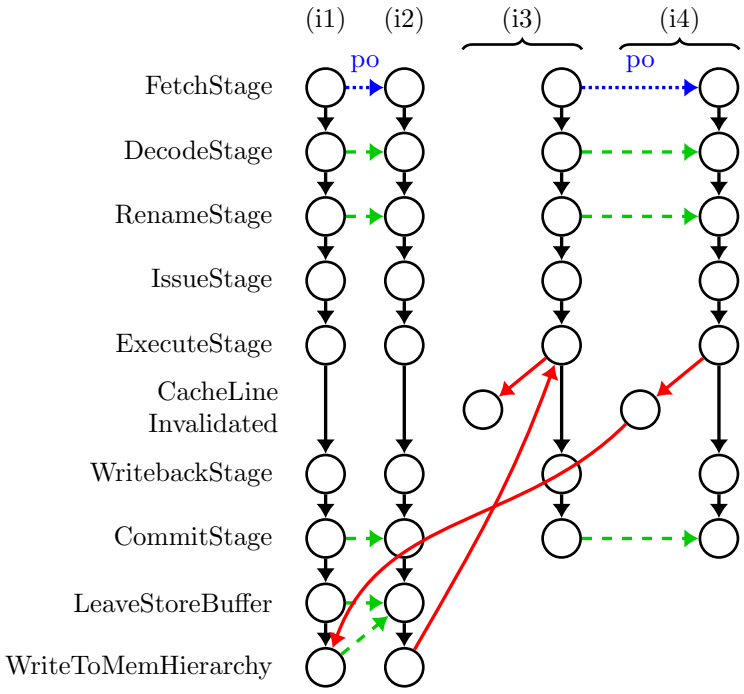
This implementation of speculative load reordering can be modeled in PipeCheck by including cache line invalidation as a "location" within the model. Figure 5.9a

shows an example of PipeCheck's use of this as applied to the gem5 O3 pipeline model and to the depicted litmus test. Extra vertices have been added to represent the invalidations of the cache lines that (i3) and (i4) read from, and the observed edges in the graph have been adjusted to account for these new vertices. In particular, the cache line that (i4) reads from must have been invalidated before (i1) wrote to memory to observe the proposed result.

PipeCheck uses this μhb graph to analyze the correctness of speculative load reordering. This is an out-of-order pipeline, so there is no sequence of edges guaranteeing $\texttt{ppo}_{arch}$ as there was in Figure 5.2b. However, although a μhb version of this edge would be sufficient, speculative load reordering proposes that it is not strictly necessary. Should the processor guarantee that the $\texttt{ppo}_{slr}$ edge is enforced instead, this would be sufficient to prevent the forbidden outcome from being observed regardless of the presence of the $\texttt{ppo}_{arch}$ edge. As a result, the core can safely realize the performance benefits of reordering (i3) and (i4) as long as it enforces either $\texttt{ppo}_{arch}$ or $\texttt{ppo}_{slr}$.

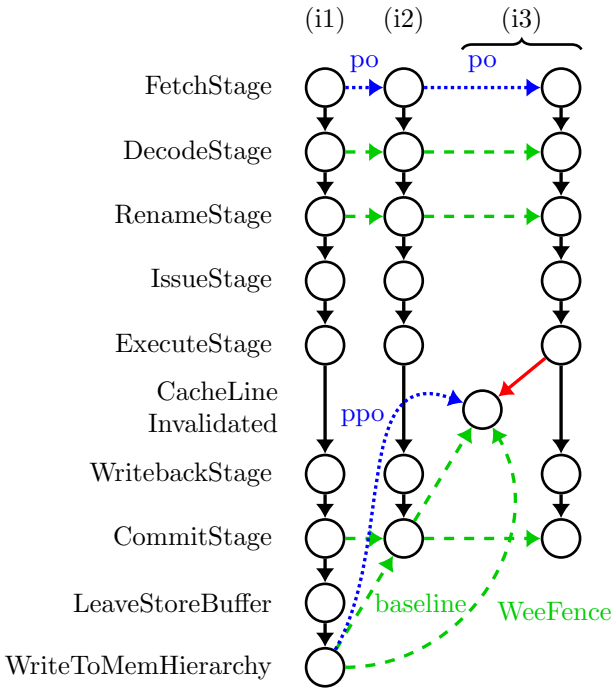
5.7.2 Consistency Bug in gem5 O3 Pipeline

For the gem5 O3 pipeline, our PipeCheck results indicated that load→load `ppo` ordering was not guaranteed, and that four of the litmus tests (including `mp`, shown in Figure 5.10a) failed validation. These results could mean either that we omitted a critical set of non-local edges in the PipeCheck definition of the pipeline, or that PipeCheck had in fact found a bug in the implementation. To analyze further, we wrote a microbenchmark to execute `mp` in a tight loop. With this, the software was in fact able to observe the forbidden result, clearly indicating that PipeCheck had found a bug.



(a) Pipeline bug shown via the `mp` litmus test. The lack of a cycle indicates that the behavior is (erroneously) observable.

While difficult to find without PipeCheck, the load→load ordering bug is easily correctable in this case[4]. The pipeline already does correctly implement load→load ordering in some cases: it squashes and restarts the second of the two reordered loads if the core sees an invalidate (as described in Section 5.7.1) to the line read by the second load, but *only if the accesses are to the same address*. Simply removing the second condition is sufficient to restore correctness. Fortunately, as actual ordering violations are relatively rare, we believe this fix results in only minimal performance changes in practice. This case study demonstrates the ability of PipeCheck to automatically find and identify very specific pipeline bugs and/or missing guarantees in the specification of the pipeline.



(a) WeeFence [DMT13] eliminates the slow "baseline" dependency while maintaining the necessary ordering.

5.7.3 WeeFence

Our third case study uses PipeCheck to verify the correctness of WeeFence [DMT13]. WeeFence proposes a microarchitectural optimization to make enforcement of Store→`mfence`→load orderings cheap under TSO. Specifically, they propose allowing post-fence loads to perform and retire prior to the fence itself, thereby reducing latency. We check that these continue to correctly enforce TSO in a particular implementation.

Figure 5.11a demonstrates the use of PipeCheck to validate the correctness of the WeeFence approach. Since their technique is not specific to a particular implementation, we apply it to the gem5 O3 pipeline model, as it allows out-of-order execution. In their baseline microarchitecture, the load may speculatively perform before the fence has retired, similarly to Section 5.7.1, but it may not retire until after the fence has retired. This in turn must happen after the store has written back

[4]The bug was independently fixed in revision 10149.

to memory. In other words, ordering is enforced from (i1.WriteToMemHierarchy) to (i2.CommitStage) to (i3.CacheLineInvalidate), where the last segment is enforced by squashing (i3) if necessary. They then propose the optimization of buffering or bouncing invalidates rather than monitoring for them, which in turn allows the read to safely retire non-speculatively, even before the store has written back to memory. This approach also enforces (i1.WriteToMemHierarchy) $\xrightarrow{\mu hb}$ (i3.CacheLineInvalidate), but without the slow intermediate step of (i2.CommitStage), thereby saving latency. This analysis demonstrates how PipeCheck can be used to verify and then to demonstrate the correctness of a microarchitectural optimization proposal.

5.8 Related Work

Shasha and Snir [SS88] and Collier [Col92] provided early frameworks to analyze programs running on machines with memory models weaker than sequential consistency. Graph-based axiomatic memory models are now widely used in academia and industry to model various architeture-level memory models of theoretical and practical interest [AAS03, Dig92, Alg12, AFI+09, AMSS10, AM06, Int13b, SPA94b, MHMS+12, YGLS03]). Most of these approaches do not model the details of any one particular pipeline; in fact, most intentionally abstract away microarchitecture-specific details in favor of presenting an implementation-independent model suitable for use at the architecture level. That being said, some of these models do (by necessity) incorporate abstracted versions of features such as branch prediction, out-of-order execution, and speculation, even though they may not all be strictly related to memory ordering, as the memory model in some cases inherently depends on the behavior of these features [AMT14, SSA+11]. In contrast, PipeCheck takes the opposite approach: it intentionally does aim to capture microarchitecture-specific features, with

the intention being to verify their correctness with respect to these more abstract architecture-level models.

Modern specifications for programming languages such as C11/C++11 [BA08, ISO11b, ISO11a] and Java [MPA05] also use axiomatic specifications as well. Software models are multiple layers of indirection away from any given hardware implementation, and it remains difficult to this day to expose hardware-level features such as dependency-based ordering enforcement in a sufficiently abstract way that they can be incorporated into general-purpose software models [MRP+14]. Due in part to this additional complexity, software models such as C++ cannot easily be incorporated into even fully general-purpose hardware memory model specification frameworks such as `herd` [AMT14].

In contrast to the static approach of PipeCheck, some authors have proposed dynamic consistency model verification techniques which incrementally build and analyze happens-before graphs [CLN03, CMP08, MS05, RZFH06]. These approaches require some degree of microarchitectural awareness, as the structures used to dynamically observe instructions must be placed at suitable points in the pipeline and/or cache hierarchy. However, in most cases they simply observe the orderings produced by the pipeline; they do not in general explicitly model how the pipeline created such an ordering, or whether other orderings may or may not be possible. Nevertheless, PipeCheck is able to model the use of dynamic tracking to enforce the consistency model rather than just to verify its correctness. In other words, PipeCheck can be used to statically verify the correctness of the dynamic enforcement mechanism.

The computational complexity of memory model analysis is largely prohibitive on the surface. Gibbons and Korach demonstrated the NP-completeness of verifying sequential consistency and linearizability [GK92, GK94, GK97]. Alur et al. showed that formally verifying the correctness of an implementation of sequential consistency is in general undecidable [AMP96]. However, as with many such for-

mally intractable problems, a great deal of use cases can be successfully analyzed in practice. Authors have demonstrated the ability to successfully use model checking [AKT13, Qad03], SAT solving [TVD10] and/or through formalization using proof assistants such as Coq [The04] or HOL [SN08] in spite of the theoretical limitations. Likewise, the PipeCheck approach benefits from small constant factors that keep its runtime tractable in spite of the complexity of verification.

Much of the effort in attempting to verify microarchitectural *implementations* of consistency models has focused on creating and evaluating litmus tests [AMSS10, AMT14, SSA+11, TQB+98], including those described explicitly in vendor specifications [AMD13, ARM09, Dig92, IBM13, Int13a, SPA94b]. Hangal et al. made an explicit effort to track down the microarchitectural sources of bugs found by testing [HVML04]. However, their approach relies on explicit debugging performed by the user, and it assumes that bugs will be found through testing. PipeCheck, in contrast, uses static analysis and automates the microarchitecture-level analysis process.

Lastly, researchers have developed certifiably-correct embeddings of certain computation patterns down to the register transfer level (RTL). Examples of circuits verified in this way include state machines, a stack machine, and even a small single-core processor [BJK+06, BC13, CGJ04, SOIG07]. More recently, Vijayaraghavan et al. have proven the correctness of a shared-memory multiprocessor written using BlueSpec [Inc04, VCAD15]. PipeCheck does not aim to replace such RTL-level formal verification; it instead aims to serve both processors which are in earlier design stages and processors which are (as of the moment) too sophisticated for RTL-level verification to be feasible, including most industry-strength processors.

5.9 Chapter Summary

We presented PipeCheck, a methodology and tool for verifying the correctness of a microarchitecture with respect to its consistency model. PipeCheck demonstrates the practicality and tractability of defining microarchitectures in terms of their location-by-location ordering properties and then verifying the correctness of their implementation of the given consistency model. Our techniques complement other ongoing efforts to verify the memory ordering correctness of various synchronization primitives and data structures from the programming language level down to the microarchitecture. We hope that in the future, PipeCheck will serve both as a framework in which designers can define their microarchitectures and as a tool by which they can verify the correctness of their implementations. PipeCheck is open-source and is publicly available at `http://github.com/daniellustig/pipecheck` [Lus14].

The next chapter extends PipeCheck to verify correct consistency enforcement in the presence of non-idealized caches.

Chapter 6

CCICheck: Verifying the Coherence-Consistency Interface[1]

While the previous chapter focused on verifying how the pipeline contributes to consistency model enforcement, this chapter presents CCICheck, an extension of PipeCheck to account for the role cache coherence protocols play in enforcing consistency models.

6.1 Introduction

In real systems, enforcement of consistency models is a collective effort. The previous chapter focused on the aspects of consistency models enforced by the pipeline, and it did so while assuming an idealized form of memory. This chapter moves towards more detailed models of consistency-related aspects of particular cache hierarchies and coherence protocols. We refer to this extended version of PipeCheck as CCICheck, where "CCI" stands for the *coherence-consistency interface*. CCICheck shows that coherence protocols can be analyzed using PipeCheck and μhb graphs as well, and in

[1]Some of the work in this chapter was performed in collaboration with fellow graduate student Yatin Manerkar and other collaborators [MLPM15].

turn, it shows how PipeCheck can be used to analyze incoherent caches, caches which occasionally return stale data, and caches which implement lazy forms of coherence.

In an abstract sense, coherence protocols and consistency models are often specified and verified independently [CGH+93, McM01, ZBES14, ZLS10]. The coherence protocol makes guarantees which are independent of the consistency model—guarantees such as the *single writers/multiple readers* (SWMR) invariant. Likewise, consistency models are defined assuming the existence of some abstract notion of "coherence". However, as discussed in Section 2.1.3, the word coherence is often overloaded. Most often, consistency models assume only write serialization: the existence of a total order on all stores to the same address.

In practice, however, the line between coherence and consistency is often blurred for the sake of aggressive performance optimizations such as speculative load reordering [AMD13, GGH91, Int13a, SPA94b]. This tight coupling drastically increases the challenge of verifying correct overall system operation. Consider the following examples. Coherence protocols may enforce write serialization, but they may also enforce additional properties such as multi-copy atomicity. This means that even though multi-copy atomicity is a consistency property rather than a coherence property, it may be enforced by the coherence protocol rather than by the pipeline. Likewise, as will be described in this chapter, some coherence protocols operate in ways that violate abstractions such as the "from-reads" edges used in the analysis of Chapter 5, and the pipeline must be aware of such behaviors when enforcing consistency in the presence of such protocols. This blurring of responsibilities indicates that verification of correct enforcement of consistency models requires more that just the pipeline models of the previous chapter. It shows that such verification also requires precise microarchitecture-level models of the ordering properties enforced (or not) by a particular cache hierarchy.

A key contribution of this chapter is that it adds much-needed precision to the definition of the interface between the coherence protocol and the consistency model. We focus in particular on the role specific coherence protocols play in enforcing consistency at the microarchitecture level. We define as the *coherence-consistency interface* (CCI) the set of memory ordering guarantees that the coherence protocol provides and that the rest of the microarchitecture expects the coherence protocol to provide. Due to the common decoupling of coherence and consistency at higher levels of abstraction, precise yet general-purpose models of the CCI did not previously exist. Empirically, this has led to much confusion about how coherence and consistency interact, and it has led to verification of such properties being an under-appreciated problem in the community.

This chapter presents CCICheck, an extension to PipeCheck to model the effects of a coherence protocol on the consistency model. Just as with PipeCheck, CCICheck models the set of orderings enforced by a coherence protocol as axioms specifying self-contained (and therefore more easily verifiable) sets of behaviors. It also introduces the concept of a ViCL, or "value in cache lifetime", which loosely represents the period of time within which a given value in a cache line can be used by an instruction in a pipeline. ViCLs allow CCICheck to model cache occupancy and coherence protocol events of interest, including the demand fetching of a line, the behavior of partially incoherent and lazily coherent memory hierarchies, and other relevant scenarios widely seen in today's systems.

CCICheck shows not only that can the CCI be formally defined; it also shows how the PipeCheck framework of Chapter 5 can also make verification tractable for microarchitectures with complex CCIs. The CCICheck approach replaces the *reads from* (`rf`), *write serialization* (`ws`), and *from-reads* (`fr`) edges used in the previous chapter with CCI-aware edges that represent microarchitectural enforcement of relationships such as SWMR by a particular coherence protocol. This finer-grained level of detail,

combined with the concept of ViCLs, helps CCICheck maintain a level of abstraction that balances generality, portability, and tractability. The key overall contribution of CCICheck is that it achieves scalable CCI-aware verification by clearly enumerating how implementation-level guarantees provided by the pipeline, the coherence protocol, and the rest of the microarchitecture combine to enforce all of the requirements of the consistency model.

The rest of this chapter is organized as follows. Section 6.2 presents a more in-depth motivation. Section 6.3 introduces the ViCL abstraction, and Section 6.4 describes how they are used in μhb graphs. Section 6.5 presents the experimental methodology, and Section 6.6 demonstrates some case studies following this methodology and some performance numbers to show that CCICheck analysis remains tractable. Lastly, Section 6.7 presents related work, and Section 6.8 concludes.

6.2 Motivating Example

Using examples of CCI interactions that need verifying in real implementations, this section motivates and introduces some key characteristics of CCICheck.

6.2.1 Coherence-Consistency Interface Mismatches

During the analysis of the previous chapter, some relationships (e.g., `ppo`) were decomposed into sets of more microarchitecturally-inspired μhb edges. Others (e.g., `rf`, `ws`, and `fr`) were interpreted in a way more directly matching the architecture-level abstractions. A key insight of this chapter is that in some cases, `rf`, `ws`, and `fr` are also too abstract to properly model the behavior of some memory hierarchies, and these edges must also therefore be decomposed into smaller components in the same way as was done for `ppo`. Since `rf`, `ws`, and `fr` edges come about due to the interaction of memory-accessing instructions in the pipeline with the caches they access, CCICheck

$$\text{acyclic}\,(\texttt{rfe} \cup \texttt{ws} \cup \texttt{fre} \cup \texttt{ppo} \cup \texttt{mfence})$$

$$\text{acyclic}\,(\texttt{rf} \cup \texttt{ws} \cup \texttt{fr} \cup \texttt{po-loc})$$

Figure 6.1: Axiomatic specification of the TSO memory model [Alg12] (reproduced from Figure 5.1b)

focuses on specifying the behavior of a given cache coherence protocol in a way that can be incorporated directly into μhb graphs. This allows coherence protocol μhb edges to serve as the microarchitecture-level replacements for the architecture-level `rf`, `ws`, and `fr` hb edges.

Consider the definition of TSO given in Figure 6.1. TSO analysis searches for cycles among edges such as `rf`, `ws`, and `fr`, among others. Unfortunately, coherence protocols do not generally specify correctness or behaviors in terms of these edge types directly. They instead verify properties such as the Single Writers/Multiple Readers (SWMR) property, which states that if a line containing address a is writable, there may be no other valid lines containing a, and the Data Value Invariant (DVI), which states that the value held by any valid line is the value written by the most recent store to that address [SHW11, SSH$^+$13, ZBES14]. While the `ws` property is implicit in these two properties, the `rfe` and `fre` properties (and the fact that they are used to check for cycles in TSO) are *stronger* assumptions than SWMR and DVI inherently provide. This CCI mismatch between the guarantees provided by the coherence protocol and the guarantees expected by the rest of the microarchitecture can lead to consistency violations if not carefully addressed.

In particular, the definitions of `rfe` and `fre` (and their use in cycle checking) implicitly assume multi-copy atomicity, which is *not* a universal property of coherence protocols, and it is *beyond* what SWMR and DVI inherently provide. In scenarios where multi-copy atomicity is not enforced by the coherence protocol, enforcement falls to the pipeline instead. The previous chapter verified pipeline correctness under

the assumption that multi-copy atomicity is being provided "for free" by the memory hierarchy, even though it is not a universal coherence protocol guarantee. If the coherence protocol were replaced with another which does not enforce multi-copy atomicity, the consistency model would be broken, and outcomes such as that in the `wrc` litmus test of Figure 3.3 would become illegally observable. This example demonstrates how at an implementation level, the pipeline and the coherence protocol must 1) have a well-defined specification of the ordering properties that they do (or do not) enforce, and 2) combine to enforce the overall consistency requirements of the architecture.

Unfortunately, there are a number of important subtleties which can arise when trying to precisely specify the consistency-relevant orderings enforced by a coherence protocol. For instance, in a protocol such as the recently-proposed TSO-CC in which coherence is enforced lazily, L1 cache lines in shared states are neither tracked nor invalidated by the L2 cache [EN14]. It is instead explicitly the job of the pipeline to invalidate shared lines in L1 caches. This implies that the coherence protocol itself does not provide strict multi-copy atomicity, and hence that the approach of the previous chapter would be insufficient to verify TSO-CC. We discuss TSO-CC further in Section 6.6.2.

6.2.2 The Window of Vulnerability Problem

As an even more subtle example of non-intuitive CCI behavior, consider the *window of vulnerability* problem, a situation in which certain coherence protocols are prone to livelock due to repeated invalidation-before-use of data propagating through a cache hierarchy [KCA92]. High-performance split-phase cache coherence protocols allow for a delay between the sending of a request and the arrival of a response, and they allow for other messages to become interleaved in between the two. Whenever this causes

a conflict, the coherence protocol is responsible for detecting the situation. It then should either cancel and retry the request or tell the pipeline to do so.

Figure 6.2 demonstrates three operational execution sequences of the `mp` litmus test. As discussed in previous chapters, under TSO, the `mp` test outcome `r1=1, r2=0` is forbidden, as neither the stores nor the loads may be reordered. The executions of Figure 6.2 begin identically, as denoted by Steps 1-10 which are common to each: a prefetch or speculative request for `x` is issued (step 1) before the load of `y` executes. The prefetched line is invalidated (step 4) by core 0's store to `x` before the data is received. The demand request for `x` from the core is also issued (step 10) before the (now stale) data for `x` arrives.

At step 11, the sequences diverge in behavior. In the first execution, the stale data is dropped and the load is retried. This maintains the TSO requirements, but no forward progress has been made, and in fact the entire sequence is prone to being repeated indefinitely, a situation known as livelock [KCA92]. In the second execution, when the stale value of `x` arrives at core 1, the coherence protocol returns that value of `x` to the core (Step 11b) in an effort to prevent livelock situation described above. When it does so, it creates a consistency violation by allowing the forbidden outcome `r1=1, r2=0` to occur. This is known as the "Peekaboo" problem [SHW11].

One solution which avoids livelock while also satisfying TSO ordering requirements is to allow access to invalidated data if and only if the accessing instruction was the oldest unperformed load or store in program order at the time the coherence request for the now invalidated data was issued [SHW11]. This effectively reorders the load to effectively have executed at the coherence point, but the extra oldest-in-program-order constraint ensures that this does not cause a consistency violation. It also ensures that forward progress will be made and hence that livelock will be avoided.

The above example clearly depicts a case where a feature of the coherence protocol (the livelock-avoidance mechanism) affects and is affected by the MCM implemen-

Core 0	Core 1
(i1) [x]←1 (i2) [y]←1	(i3) r1←[y] (i4) r2←[x]
Under TSO: Forbid r1=1, r2=0	

(a) Code for Litmus Test `mp`

	Core 0	Core 1
1		x: prefetchS miss, issue GetS/IS^D
2	x: receive Fwd-GetS, send Data[0]/S	
3	x: store miss; issue GetM/SM^{AD}	
4	x: receive Data[0](ack=1)/SM^A	x: receive Inv, send Inv-Ack/IS^{D}I
5	x: receive Inv-Ack, perform store/M[1]	
6	y: store hit/M[1]	
7		y: load miss, issue GetS/IS^D
8	y: receive Fwd-GetS, send Data[1]/S[1]	
9		y: receive Data[1], perform load r1=1/S[1]
10		x: load miss, stall/IS^{D}I
11a		x: receive and drop Data[0], replay GetS/IS^D
12a	x: receive Fwd-GetS, send Data[1]/S	
13a		x: receive Data[1], perform load r2=1/S[1]

(b) The Baseline WoV solution drops stale data upon receipt: livelock-prone, but no consistency violation

	Core 0	Core 1
11b		x: receive Data[0], perform load **r2=0**/I

(c) If livelock avoidance is naively added for WoV cases, steps 11a, 12a, and 13a are replaced simply by step 11b. Stale data returned, resulting in a consistency violation.

	Core 0	Core 1
11c		x: receive Data[0], but drop it because there was another load (step 7) between the coherence request (step 1) and this load (step 10); replay GetS/IS^D
12c	x: receive Fwd-GetS, send Data[1]/S	
13c		x: receive Data[1], perform load r2=1/S[1]

(d) A livelock-free solution to the Peekaboo problem is to return invalidated data if and only if the load (step 10) was the oldest in program order at the time the coherence request (step 1) was issued [SHW11]

Figure 6.2: Three executions of `mp` using different coherence protocols

tation in a way that goes beyond traditional coherence protocol properties such as SWMR. This particular case affects the `fre` edge of Figure 6.1: it effectively states that at an implementation level, `fre` does not hold whenever the coherence protocol returns already-invalidated data. Just as the previous chapter described sets of μhb edges which indirectly replace `ppo`, there must be a set of μhb edges which indirectly replace `fre` during the Peekaboo situation. This in turn requires the μhb graph to explicitly account for of the behavior of the coherence protocol and for events such as the "original coherence request" used in the example above.

6.3 The ViCL Abstraction

This section introduces the Value-in-Cache-Lifetime (ViCL) abstraction. ViCLs provide the abstraction by which cache occupancy and value propagation can be modeled and verified using μhb graphs. The ViCL abstraction can track coherence events relevant to memory ordering while remaining sufficiently abstract that the PipeCheck verification approach remains scalable and fully general.

6.3.1 ViCLs: Definition and Usage

Conceptually, a ViCL represents the period of time (relative to a single cache) over which a given value[2] is present in a specific cache or memory. To formally define a ViCL, we first conceptually assign a unique *cache_id* to every cache in the system, and a unique *generation_id* to each line brought into a given cache over the duration of an execution. This allows us to uniquely refer to each cache line in an execution

[2]We assume without loss of generality that each store produces a unique value even if its contents are identical with the value of another store. This does not affect correctness, as our solver would consider either value to be a possible source for a load returning the value in question.

using its *cache_id* and *generation_id*. Formally, a ViCL is a 4-tuple

$$(cache\_id, address, data\_value, generation\_id)$$

which maps onto the period of time within which the cache line corresponding to *generation_id* in the cache identified by *cache_id* holds the value *data_value* for address *address*. A given address and data value pair may have many matching ViCLs over the course of an execution. These ViCLs could be in different caches (different *cache_id*s), be brought into the same cache at different points in the execution (different *generation_id*s), or both. Likewise, changes in data values also cause ViCL changes, which is key to their use in enumerating possible read-write pairings for consistency verification. There may also be gaps between ViCLs for a given address and cache pair where there is no value in the cache for that address. In addition, our definition allows for the (admittedly uncommon, but feasible) possibility that a cache may hold two lines for the same address simultaneously.

A ViCL time period starts at a *ViCL Create* event and ends at a *ViCL Expire* event. ViCL Create and ViCL Expire events represent the points in time at which the corresponding ViCL 4-tuple either starts or stops serving the data in question. A ViCL Create event occurs for address `x` when either (i) a cache line containing `x` enters a usable state from a previously unusable state, or (ii) when a value is written to `x` in a cache line. A ViCL Expire event for address `x` occurs when (i) its cache line enters an unusable state from a previously usable state, or (ii) a value is written to `x` in a cache line.

ViCLs are finer-grained than cache lines, as cache lines hold more than one address. Note that the creation and expiration of a ViCL for a given address has no inherent effect on ViCLs of other addresses, even if they do share a cache line. Any such sharing

possibilities (e.g., evictions due to false sharing) are already handled by CCICheck's comprehensive enumeration. In such cases, our analysis is conservative but correct.

We can also assign a unique *cache_id* to capture all of main memory. Because memory lines are not evicted in the same way that cache lines are, the *generation_id* for all main memory ViCLs will never change[3]. ViCLs representing main memory can be useful to represent uncacheable accesses.

6.3.2 ViCL Timeline Example

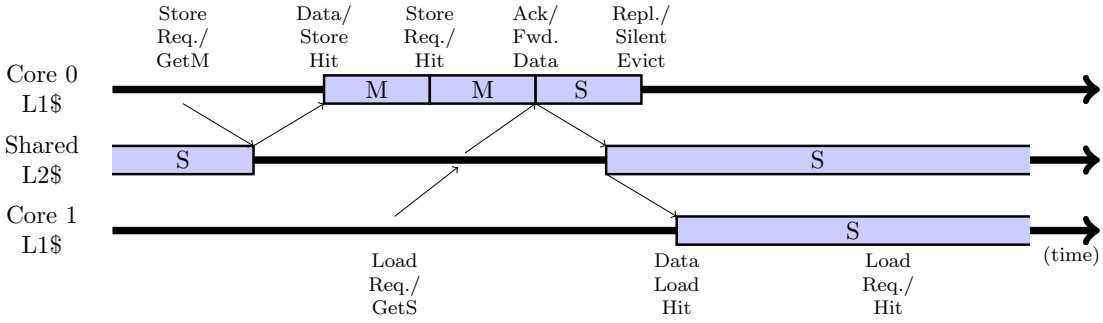
To explain the intuition behind ViCLs, Figure 6.3 shows an example of how litmus test `co-mp` (Figure 6.3a) might be executed on a processor with private L1 caches and a shared L2 cache (all of which are coherent with each other). Figure 6.3b presents a timeline using traditional notions of cache line state, while Figure 6.3c presents the same timeline using ViCLs. In addition to ViCL create and expire points, it also denotes the value of ViCL 4-tuples at noteworthy points in the timeline. At the beginning, the shared L2 cache holds the value 0 for address `x`. The first thread 0 store then misses in its L1 cache, causing it to fetch the line for exclusive ownership. When the L2 cache receives the request, the L2 cache's ViCL expires[4]. Once the data arrives at core 0's L1 cache and the store completes, a new ViCL is created at the L1 cache, representing both the move to a valid state and the writing of new data. When the second store then completes, the first L1 ViCL (which had a data value of 1) expires, and a new ViCL is created for `x` with the value 2. When thread 1 starts executing sometime later, it will fetch the data from core 0's L1 cache. This does not cause a ViCL expiration, as none of the components of the core 0 L1 ViCL's 4-tuple change. Instead, the L1 cache forwards the data to the other L1 through the L2, creating new ViCLs in those caches in the process.

[3]...unless data is swapped from memory out to disk. We do not consider such a scenario in this thesis, although the formalism supports it.

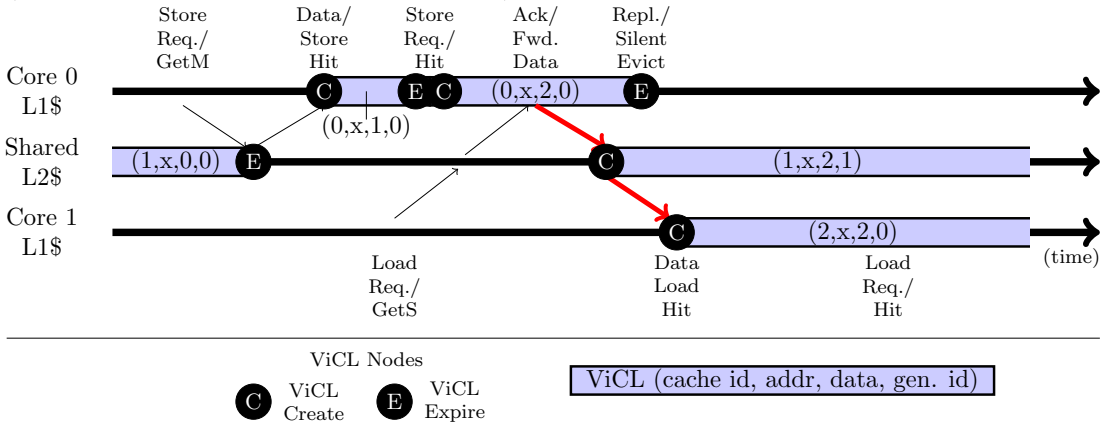
[4]The line may not necessarily be invalidated; it may move to a state tracking the L1 cache as the new owner. Nevertheless, the ViCL expires because the old data is no longer being served.

Thread 0	Thread 1
(i1) St [x]←1	(i3) Ld [x]→r1
(i2) St [x]←2	(i4) Ld [x]→r2
In TSO: r1=2, r2=2 Allowed	

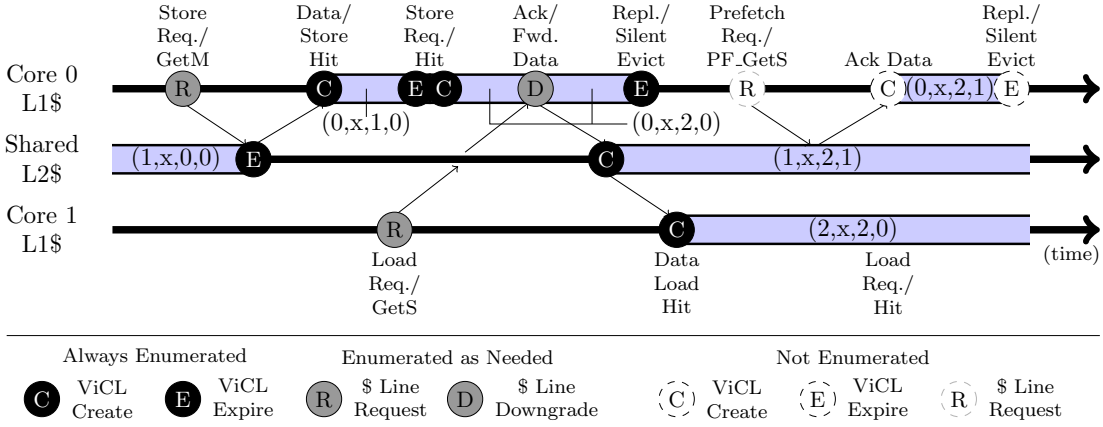
(a) Variant of litmus test `co-mp` (normally, the outcome has r2=1)



(b) Sample timeline for an execution which produces the legal outcome r1=2, r2=2. (M = Modified state, S = Shared state)



(c) Same timeline, but with ViCLs added. The thicker red arrows correspond to those in Figure 6.4.



(d) Same timeline, but with additional cache line events that may be needed to model certain scenarios

Figure 6.3: An example of how ViCL nodes relate to events in the cache hierarchy.

In some scenarios, ViCL create and expire events may be sufficient to verify all necessary orderings. In more complex scenarios, however, we can add additional coherence-related events to the timeline, and infer orderings for those events with respect to ViCL create and/or expire events. For example, a cache line downgrade event may take place between ViCL Create (for a cache line in an exclusive state) and ViCL Expire. Similarly, a cache line request event will happen before the ViCL it fetches gets created. Figure 6.3d shows how these additional events can be included into the timeline for the same execution of `co-mp`. Such additional nodes are required for our Peekaboo and TSO-CC case studies in Section 6.6.

6.3.3 Using ViCLs in μhb Graphs

A key benefit of ViCL events is that they map naturally into nodes within μhb graphs. Likewise, orderings which are enforced between ViCL events due to coherence protocol behavior map naturally onto μhb graph edges.

Each possible execution of a program (whether allowed or forbidden by a given consistency model) gives rise to a mapping from each cache-accessing instruction to the ViCL accessed by that instruction. When modeling a particular execution by a μhb graph, we add μhb nodes to the graph for every cache-accessing instruction representing the ViCL Create and ViCL Expire events for the ViCL(s) that instruction accesses in the execution. We rely on the microarchitecture definition (Section 6.4.1) to list all possible caches that a memory-accessing instruction could interact with (e.g., read directly from L1, read from L2 through the L1, etc.). The address and data of a ViCL's 4-tuple must match those of its instruction, and the possibilities for a ViCL's generation ID are used to check whether two ViCLs of the same address, value, and location are the same ViCL or not.

Figure 6.4 illustrates a CCICheck μhb graph with ViCL nodes for the execution timeline depicted in Figure 6.3c. The four ViCLs in the graph represent the four

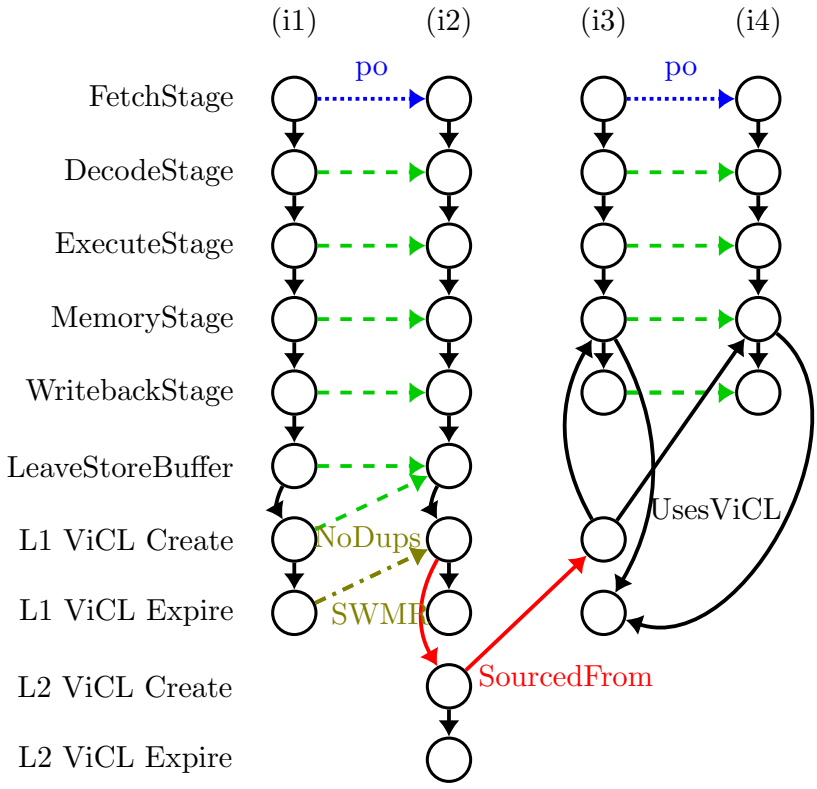


Figure 6.4: CCICheck graph for the Figure 6.3c scenario.

rightmost ViCLs in Figure 6.3c (the very first ViCL does not need to be enumerated as its data is not read by any instruction). The edges between the ViCLs reflect the ordering constraints in the microarchitecture specification (Section 6.4.1). For example, the dot-dashed brown "SWMR" edge enforces that ViCLs for the first write to `x` (i1) must expire before the ViCLs for the subsequent write (i2) are created, as per the coherence protocol's SWMR invariant. The "NoDups" edge (which in this case overlaps with the "SWMR" edge) between the two ViCLs for `x` in core 0's L1 cache enforces that the first ViCL for `x` in the cache must expire before a second one for the same address in the same cache can be created (i.e., there can be no duplicates within a single cache at any given time). This reflects a local property of each individual cache. The two red "SourcedFrom" edges correspond to the arrows in Figure 6.3c showing `x=2` being propagated from the core 0 L1 cache to the core 1 L1

cache through the L2. Finally, the four thick solid black "UsesViCL" edges represent the fact that i3 and i4 access the same ViCL in this execution.

In general, for each load, μhb edges represent the fact that the instruction must access the ViCL sometime between its creation and expiration. Likewise, for each store, μhb edges represent the fact that the ViCL is created when the store value reaches the cache (because it changes the data component of the ViCL 4-tuple). μhb edges are also used to represent any orderings enforced or implied between ViCLs due to coherence protocol behavior (such as the "SWMR" edge in Figure 6.4) or memory hierarchy restrictions (like the "NoDups" edge in Figure 6.4). The details of these orderings are specific to each protocol (or class of protocols). If other cache line events (like fetch requests and downgrades) are modeled, then the graph will include μhb nodes representing these events and edges representing their orderings with respect to other events in the graph. Examples of microarchitectures that require such modeling can be found in Section 6.6.

The maps from instructions to ViCLs for an execution are not injective (since there may be multiple instructions which map to a single ViCL); each ViCL's create and expire nodes appear in the μhb graph no more than once. They are also not surjective because there may be ViCLs which are not accessed by any instruction, such as the first ViCL in Figure 6.3c. We do not need to draw such non-accessed ViCLs as they are not relevant to consistency enforcement.

6.4 CCICheck and μhb Graphs

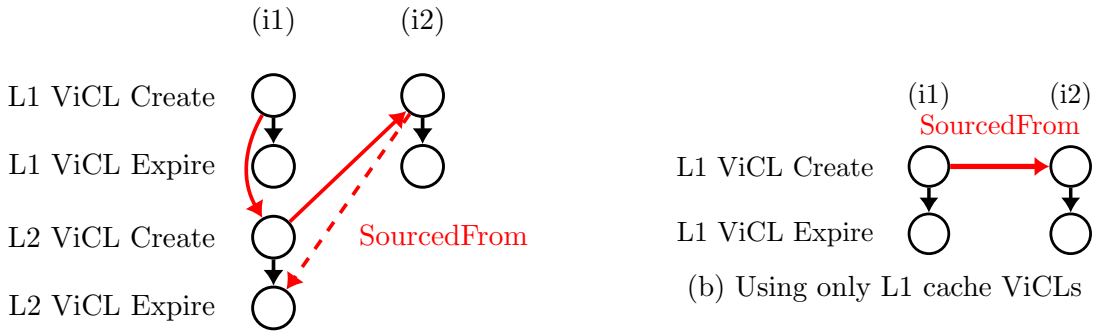
This section describes how a CCICheck microarchitecture definition specifies ordering relationships, as well as how CCICheck conducts its enumeration of μhb graphs. We begin by describing the general approach, and we give an in-depth example of how the approach is used to model a single level of caching. We then follow that with a

description of how the approach extends to models which explicitly include multiple levels of caching.

6.4.1 ViCL-Aware Microarchitecture Definitions

CCICheck microarchitecture definitions extend the approach of Section 5.3 to account for the presence of ViCLs. At a high level, models should attempt to decouple the implementation-level orderings enforced by the pipeline from the implementation-level orderings enforced by the cache coherence protocol. Note, however, that the line between the pipeline and the coherence protocol may differ from the architecture-level understanding of the same line, and the line may at times be blurry. In most cases, the pipeline definitions from Chapter 5 can be incorporated and directly reused in this context. However, the idealized `rf`, `ws`, and `fr` orderings of the previous chapter have been removed in favor of the ViCL-based approach described in this section.

On one hand, a desirable property of the coherence protocol models is that they involve self-contained axioms of the form that are (or might naturally be) verified by a coherence protocol verifier. For example, the models may define axioms representing properties such as the Single-Writer/Multiple-Readers (SWMR) guarantee and Data Value Invariant (DVI) (Section 6.2) rather than lower-level protocol implementation details. This choice serves two purposes. First, it allows us to build off of (rather than compete with) the significant research effort that has gone into verifying coherence protocols [SSH$^+$13, ZBES14, ZLS10, VV14, McM01]. Second, it (alongside our ViCL abstraction) meets our goals of explicitly modeling the orderings required to verify consistency while abstracting away any lower-level details. We expect that these details are at a similar level to properties such as SWMR and DVI, and we therefore expect that they could independently be verified in a similar manner to those properties.



(a) Explicitly using L1 and L2 cache ViCLs

(b) Using only L1 cache ViCLs

Figure 6.5: It is possible to model a system with multiple levels of caching using a microarchitecture model with only L1 cache ViCLs. Some "SourcedFrom" edges simply become more abstract, and others may simply be lost. Edges of the latter class (e.g., the dashed edge above) must be replaced by other axioms as needed.

On the other hand, at the implementation level, in contrast to what might be expected, few coherence protocol axioms are universal. While properties such as SWMR may hold true in general in an abstract sense, the implementations may vary widely. For example, while eager coherence protocols may enforce SWMR in a strict physical sense, lazy coherence protocols often enforce it only in a more abstract logical sense, and the two approaches would not behave the same when modeled in μhb graphs. We therefore leave the specific choice of axioms to each individual microarchitecture model.

6.4.2 Example: A Model with L1 Cache ViCLs Only

We start by explaining in detail a microarchitecture model which only explicitly includes ViCLs for the L1 cache. This does not inherently preclude the model from describing systems with multiple levels of cache; it simply abstracts away any communication which takes place via deeper levels of the cache. For example, two of the three thicker `SourcedFrom` edges in Figure 6.5a can be captured using the one thicker `SourcedFrom` edge in Figure 6.5b. The third edge cannot; if this edge is needed, it

```
DefineMacro "FindL1ViCL":
  exists microop "s",
    SamePhysicalAddress s i /\ SameData s i /\ SameCore s i /\
    EdgesExist [((s, ViCLCreate ), (i, MemoryStage), "SourcedFrom") /\
                ((i, MemoryStage), (s, ViCLExpire ), "SourcedFrom")].

Axiom "Reads":
  forall microops "i",
    OnCore c i => IsAnyRead i =>
    EdgesExist [((i, Fetch      ), (i, Decode     ), "path");
                ((i, Decode     ), (i, Execute    ), "path");
                ((i, Execute    ), (i, MemoryStage), "path");
                ((i, MemoryStage), (i, Writeback  ), "path")] /\
    (ExpandMacro STBFwd \/ (ExpandMacro STBEmpty /\ ExpandMacro FindL1ViCL)).

Axiom "Writes":
forall microops "i",
  OnCore c i => IsAnyWrite i =>
  EdgesExist [((i, Fetch           ), (i, Decode          ), "path");
              ((i, Decode          ), (i, Execute         ), "path");
              ((i, Execute         ), (i, MemoryStage     ), "path");
              ((i, MemoryStage     ), (i, Writeback       ), "path");
              ((i, Writeback       ), (i, LeaveStoreBuffer), "path");
              ((i, LeaveStoreBuffer), (i, ViCLCreate      ), "path");
              ((i, ViCLCreate      ), (i, ViCLExpire      ), "path")].
```

Figure 6.6: CCICheck axioms representing pipeline behavior for a processor with private L1 caches. See Figure 6.8 for the associated ViCL axioms.

must be captured by some other axiom instead. Section 6.4.3 describes how ViCLs for deeper levels of the cache can be explicitly modeled when needed.

Figures 6.6 and 6.8 present the pipeline-relevant and ViCL-relevant subsets of a relatively simple CCICheck microarchitecture model, respectively. Macros `STBFwd` and `STBEmpty`, axioms `mfence`, `RMW`, and the pipeline stage `in-order` axioms are unmodified from Figures 5.3 through 5.5 and hence are not pictured. Axioms `BeforeOrAfterEveryWriteToSameAddr` and `WriteSerialization` are replaced entirely by the cache-centric axioms in this section. We explain each of these new axioms in turn below.

Macro `FindL1ViCL` and Axioms `Reads` and `Writes`. Every cache-accessing instruction interacts with exactly one ViCL. Loads and stores behave very differently, however. Each store creates a new ViCL, since each store is assumed to write a new unique data value (Section 6.3.1) and hence instantiates a new ViCL 4-tuple. The ViCL Create and ViCL Expire μhb nodes are simply appended to the end of the

path for each store. Note that the ViCL Expire node may occur long after the events represented by the rest of the nodes in the path have taken place.

Loads require more complicated behavior, as they may or may not share ViCLs with other loads or stores in the program. Store buffer forwarding behaves identically to the implementation in the previous chapter; our explanations in the rest of this section therefore assume a store buffer miss (i.e., `STBFwd` returns false and `STBEmpty` returns true), and we focus on the implementation of `FindL1ViCL`. Every load instruction which takes a store buffer miss will read from the cache, which means that it will return the data stored in some ViCL. We refer to this as a *sourcing* relationship.

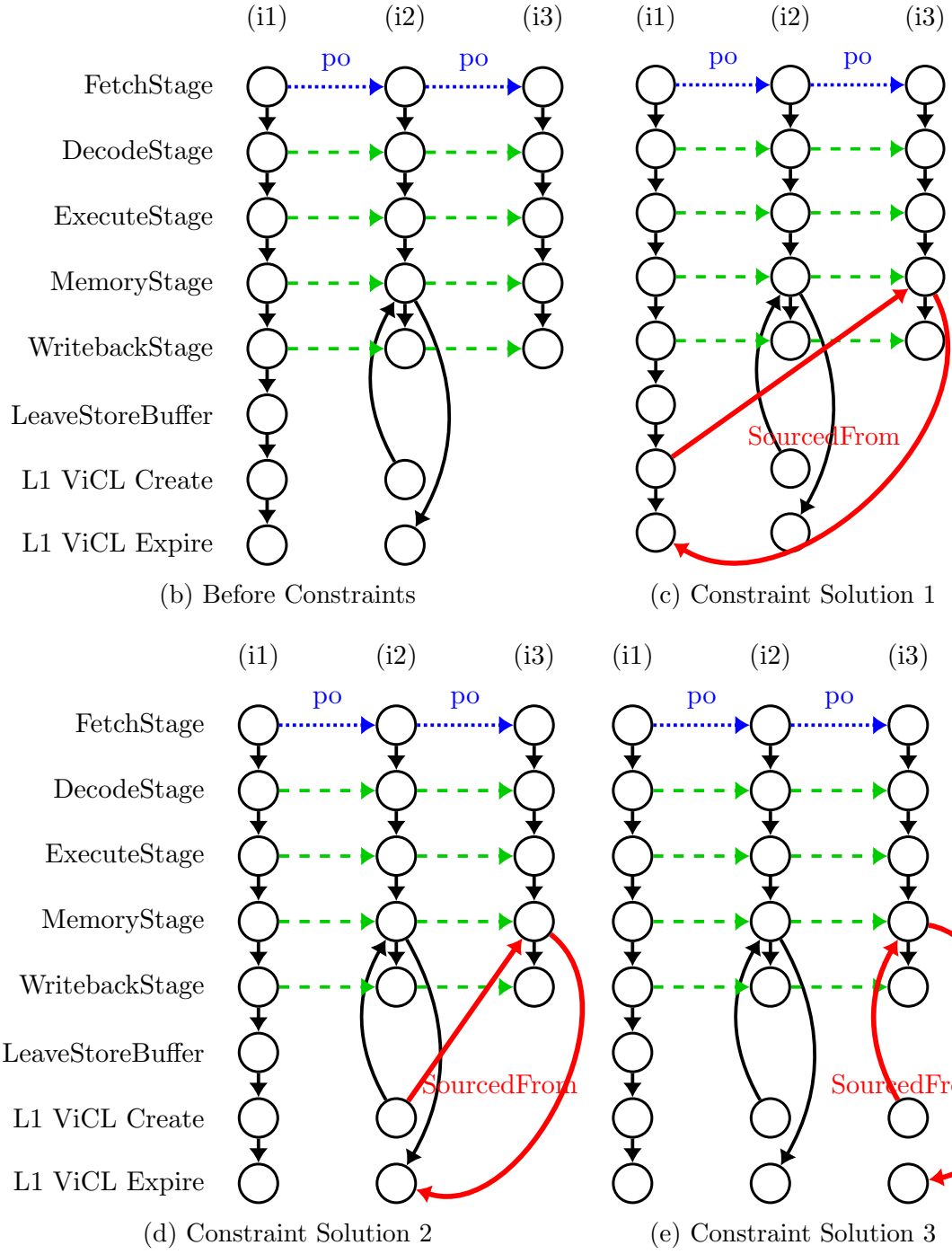
As its name suggests, the `FindL1ViCL` macro searches for a ViCL from which load instruction `i` sources its data. This macro enumerates all possible mappings between load instructions and ViCLs. It does not enumerate all concrete generation IDs; it instead only enumerates all possible ways in which instructions may or may not share ViCLs by enumerating all ways in which they do or do not share a matching generation ID. The actual generation IDs themselves are moot.

The `FindL1ViCL` macro searches for all L1 cache ViCLs which match the same cache id, address, and data. Some possible solutions may already be associated with other instructions. In this case, `i` reuses the ViCL associated with that instruction. Otherwise, the macro may instantiate a new ViCL which is not yet associated with another instruction.

Figure 6.7 depicts the `FindL1ViCL` axiom in action. Figure 6.7a shows the code evaluated in this scenario. This code is single-threaded for the sake of having legible μhb graphs, although multithreaded examples work identically. Figure 6.7b shows a partially-evaluated scenario in which (i1) and (i2) were chosen to each have their own ViCL (i.e., `s` was chosen to equal `i`). The remaining three subfigures depict the three ways in which the same axiom can be evaluated for (i3). Figure 6.7c depicts (i3) sharing a ViCL with (i1) but not with (i2); this scenario will likely be ruled out

Thread
(i1) St [x]←1
(i2) Ld r1←[x]
(i3) Ld r2←[x]
Outcome r1=1, r2=1: Allowed

(a) Sample thread



(b) Before Constraints

(c) Constraint Solution 1

(d) Constraint Solution 2

(e) Constraint Solution 3

Figure 6.7: Evaluation of macro `FindL1ViCL` for instruction (i3)

in any memory model forbidding load reordering. Figure 6.7d depicts (i3) sharing a ViCL with (i2); this is valid and even the most likely case, as (i3) would most likely benefit from the locality in the cache and take a cache hit. Lastly, Figure 6.7e depicts (i3) having its own ViCL; this is less likely than the preceding cache hit scenario, but it is possible if, for example, the thread were to be interrupted between (i2) and (i3).

As in the previous example, some instruction-to-ViCL mappings may be ruled out by coherence protocol orderings which combine to form a cycle. The coherence protocol axioms are shown in Figure 6.8. We discuss them below.

Macros `L1ViCLSource` and `L1ViCLSourceInitial` and Axiom `L1ViCLs`. Just as every load instruction which does not forward data from the store buffer must source its data from some ViCL, each ViCL in turn must have its own source for its data value. L1 cache ViCLs associated with store instructions are sourced directly from a store instruction. This is represented by the inclusion of L1 ViCL Create and L1 ViCL Expire nodes directly within the path of each store (via the `Writes` axiom). All other ViCLs must source their data from some other ViCL, thereby forming a "chain" of ViCL sourcing relationships that ultimately points back to the store that originally produced the data in question. In models with multiple levels of caching, an L1 cache ViCL would most often source its data from a ViCL in a lower-level cache. Figure 6.4 presented an example of this sourcing requirement.

The macros `L1ViCLSource` and `L1ViCLSourceInitial` and `L1ViCLs` work as follows. Axiom `L1ViCLs` asserts that every ViCL instantiated by the search process described above must have a source, and that this sourcing may take place according to one of the two following macros. Macro `L1ViCLSource` finds a candidate ViCL `i'` and adds two edges indicating that the memory access takes place while the ViCL is still valid. The latter enforces the Data Value Invariant (DVI) (Section 6.2.1). Macro `L1ViCLSourceInitial` asserts that the ViCL associated with `i` itself sources from the

```
DefineMacro "L1ViCLSourceInitial":
  DataFromInitialStateAtPA i /\
  forall microop "w",
    ~SameMicroop i w => IsAnyWrite w => SamePhysicalAddress w i =>
    EdgeExists ((i, ViCLExpire), (w, ViCLCreate), "DVI", "red").

DefineMacro "L1ViCLSource":
  exists microop "i'",
    ~SameMicroop i i' /\ amePhysicalAddress i i' /\ SameData i i' /\
    EdgesExist [((i', ViCLCreate), (i , ViCLCreate), "SourceFrom")] /\
    ~exists microop "i''",
      SamePhysicalAddress i i'' /\ IsAnyWrite i'' /\
      EdgesExist [((i' , ViCLCreate), (i'', ViCLCreate), "DVI");
                  ((i'', ViCLCreate), (i'', ViCLCreate), "DVI")].

Axiom "L1ViCLs":
  forall microop "i",
    OnCore c i => IsAnyRead i =>
    NodeExists (i, ViCLCreate) \/ NodeExists (i, ViCLExpire)) =>
    EdgeExists ((i, ViCLCreate), (i, ViCLExpire), "path") /\
    (ExpandMacro L1ViCLSourceInitial \/ ExpandMacro L1ViCLSource).

Axiom "SWMR":
  forall microops "i1",
    IsAnyWrite i1 =>
    (NodeExists (i1, ViCLCreate) \/ NodeExists (i1, ViCLExpire)) =>
    forall microops "i2",
      (NodeExists (i2, ViCLCreate) \/ NodeExists (i2, ViCLExpire)) =>
      ~SameMicroop i1 i2 => SamePhysicalAddress i1 i2 => (
        (EdgeExists ((i1, ViCLCreate), (i2, ViCLCreate), "swmr")) \/
        (EdgeExists ((i2, ViCLExpire), (i1, ViCLCreate), "swmr"))).

Axiom "L1ViCLNoDups":
  forall microop "i1",
    (NodeExists (i1, ViCLCreate) \/ NodeExists (i1, ViCLExpire)) =>
    forall microop "i2",
      ~SameMicroop i1 i2 => SameCore i1 i2 =>
      (NodeExists (i2, ViCLCreate) \/ NodeExists (i2, ViCLExpire)) => (
        EdgeExists ((i1, ViCLExpire), (i2, ViCLCreate), "NoDups") \/
        EdgeExists ((i2, ViCLExpire), (i1, ViCLCreate), "NoDups")).
```

Figure 6.8: CCICheck axioms representing memory hierarchy behavior for a processor with private L1 caches, the vanilla Single Writer/Multiple Readers (SWMR) axiom, and the Data Value Invariant (DVI). See Figure 6.6 for the associated pipeline axioms.

(unpictured) initial condition, and that this implies the additional constraint that no other ViCL of the same cache and address has been created at that point.

Axiom `SWMR`. Axiom `SWMR` describes a straightforward implementation of the Single Writers/Multiple Readers axiom. It states that for any pair `i1` and `i2` of ViCLs accessing the same address, if `i1` is a write, then either `i2` must have been invalidated prior to the creation of `i1`, or `i2` must be created after `i1` is created. The former represents eager invalidation of cache line sharers, while the latter indicates that subsequent cache lines may either be sharers of the same data or may be events happening even later in the timeline. Note that the cache line serving the ViCL `i1` may be downgraded to allow for sharers (i.e., "multiple readers"), but this does not cause the ViCL to expire. Section 6.6.2 returns to subtleties such as lazy invalidation of sharers and cache line downgrade events.

Axiom `L1ViCLNoDups`. The `L1ViCLNoDups` axiom defines a relatively straightforward property: that in a given cache, at a given time, there cannot be more than one valid cache line holding a single address. It is technically possible for this assumption to be violated. Virtually-indexed virtually-tagged caches may indirectly fall victim to this due to the presence of synonyms (i.e., two virtual addresses which point to the same physical address), although VIVT caches are uncommon in no small part to avoid this problem. We assume that `L1ViCLNoDups` (and its counterpart `L2ViCLNoDups`) hold in all microarchitectures modeled in this thesis, but since it is an axiom specified using the DSL of Section 5.3.2, we could easily remove it to test scenarios with synonyms and potential duplicates.

6.4.3 Multilevel Caches

Although Figure 6.7 depicts the model of a single-level of caching, CCICheck models naturally adapt to systems with multiple levels of private and/or shared caches. The approach is generally the same as the approach for single-level models, except that

Name	Cache Hierarchy	Protocol Classification
SharedL1	Shared L1	Eager
PrivL1	Private L1s	Eager
Peekaboo	Private L1s	Eager, with Livelock Prevention
PrivL1/SharedL2	Private L1s, shared L2	Eager
TSOCC	Private L1s, shared L2	Lazy

Table 6.1: Memory hierarchy specifications and coherence protocol features of the microarchitectures analyzed in this chapter

multi-level cache models must be more precise about differences between levels of the hierarchy. For example, a model must specify whether L1 ViCLs may or may not source data directly from the L1 caches of other cores, or whether any such communication must pass through the L2 cache. It must also specify whether L1 cache bypassing is allowed, whether caches are write-back vs. write-through, and so on. All of these properties can be expressed directly within the PipeCheck DSL.

In many cases, the correctness of data propagation through the hierarchy can be left to the independent coherence protocol verification process. Lower-level caches need to be modeled only when some aspect of their behavior is critical to the enforcement of the consistency model. The Figure 6.7 model of an L1 cache "zooms in" on the idealized model of the previous chapter in order to provide enough detail to model situations such as the Peekaboo problem. For simple cases, this zooming may be unnecessary, but it is still correct. In more complex cases, this zooming may in fact be necessary. Likewise, certain protocols may require the model to "zoom in" even further to capture behaviors specific to the L2 cache as well. Section 6.6 will provide examples.

6.5 Experimental Methodology

Table 6.1 describes the various microarchitectures that we analyze in this section. In order to emphasize how unexpected coherence protocol behaviors can appear even in

simpler microarchitectures with in-order cores, we model the architectures as having a five-stage in-order pipeline model. Nevertheless, CCICheck easily adapts to more complex and/or less restrictive pipelines.

Our models cover a variety of memory hierarchies and coherence protocols, including single and dual layers of private and/or shared caches. The "Eager" coherence protocol classification refers to an abstract vanilla coherence protocol that eagerly invalidates sharers before every write and which is adapted to each different cache hierarchy arrangement. Other classifications listed in Table 6.1 reflect those described in Section 6.6.

The analysis is performed using the tool and methodology of Section 5.4 and the suite of x86-TSO litmus tests from Section 5.5. As Section 6.6.4 shows, although the μhb graphs are larger in this chapter due to the more sophisticated coherence protocols, the performance of our automated tool remains acceptable in practice. Interestingly, at various times during the development process, a pipeline model would pass all of the tests in the suite, yet we would later discover that certain corner cases were not addressed. Whenever we became aware of such situations, we added hand-written litmus tests to stress coherence protocol-inspired features that were not covered by the existing suite. This highlights the inherent limitations of litmus test-based approaches, and it indicates the value of more complete formal proofs of the kind discussed in Section 5.4.6.

6.6 Case Studies

The previous section presented the general approach to modeling ViCLs using CCICheck. This section presents some more advanced case studies which model coherence protocols of the kinds that are used in real-world high-performance processors.

6.6.1 Partially-Incoherent Caches and L1 Cache Bypassing

Our first example studies extremely weak memory models such as those found in many GPU systems today [NVIa]. GPUs have very weak default memory models, so their consistency enforcement tends to come in the form of explicit fences rather than preserved program order. GPUs also generally claim to be coherent, for some (often unspecified) definition of coherence (cf. Section 2.1.3) [NVI13b]. In spite of this, Alglave et al. recently found that coherence requirements are often violated in practice [ABD+15]. CCICheck and ViCLs provide a natural framework in which the consistency implications of such scenarios can be analyzed.

Alglave et al. tested GPUs using (among other tests) a modified version of the `mp` litmus test. As Figure 6.9a shows, this version of `mp` has `membar` fences inserted between the two stores and between the two loads. Since GPUs are otherwise allowed to reorder freely, the fences should prevent any reorderings, which should in turn prevent the forbidden outcome `r1 = 1, r2 = 0`. However, they discover empirically that the forbidden outcome *is* incorrectly observable on many modern GPUs. As the studied microarchitectures are proprietary, we apply CCICheck to a hypothetical yet realistic GPU model with small, in-order cores and private L1 caches following the no-allocate-on-write policy (i.e., on an L1 cache miss, bypass the L1 cache entirely).

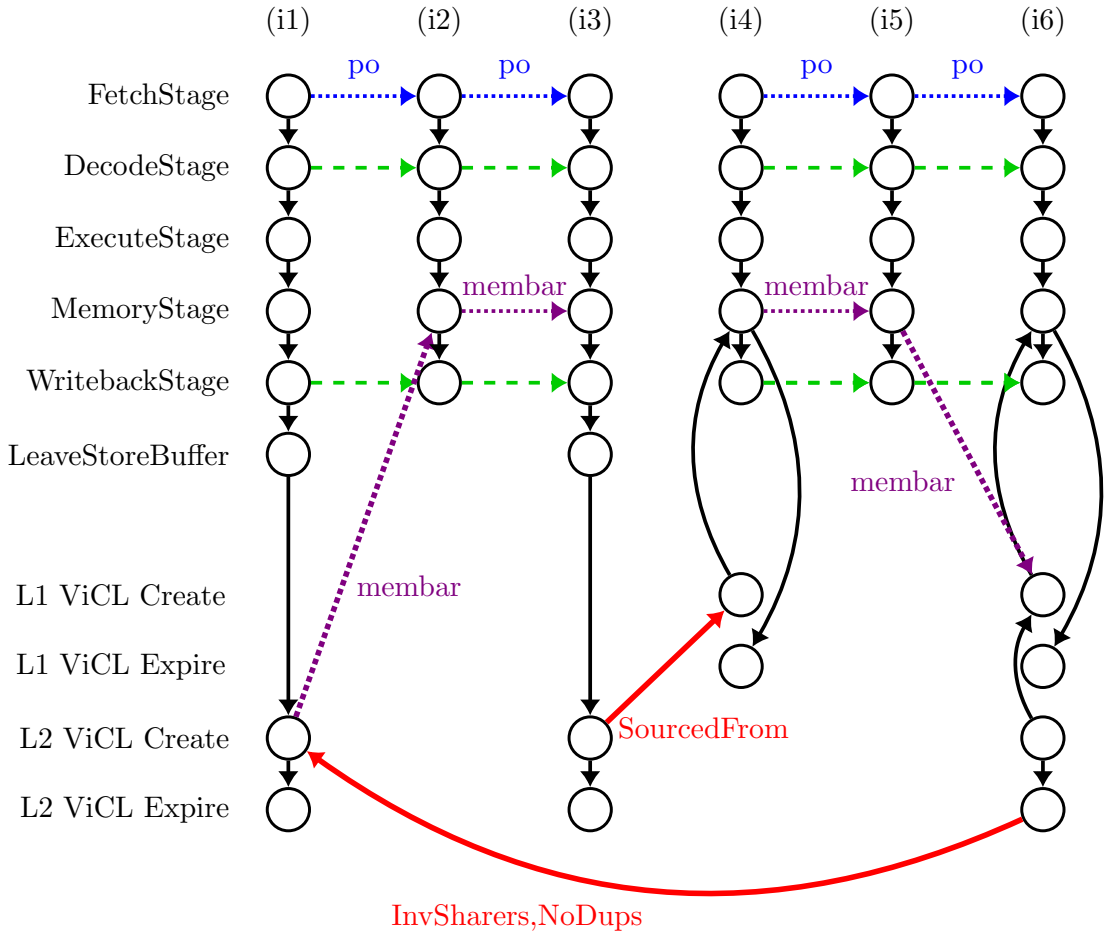
Figure 6.9b depicts a CCICheck graph for the litmus test of Figure 6.9b. Both stores write directly to the L2 cache, while both loads read from the L1. The L1 ViCL for (i6) can only have the test's prescribed value of 0 if it reads its value from the L2 before the (i1) store of `x` reaches the L2 cache. Unfortunately, in this scenario, there is no cycle in the graph to prevent the bad outcome from happening.

One explanation for this hypothetical scenario is that there is a subtle race condition which allows the (i6) L1 ViCL to be created after the fence (i5) has retired but sourced from an L2 ViCL which had been invalidated prior to the fence retired. GPUs contain complicated throughput-optimizing buffers and networks-on-chip be-

Thread 0		Thread 1	
(i1)	[x]←1	(i4)	[y]→r1
(i2)	membar	(i5)	membar
(i3)	[y]←1	(i6)	[x]→r2

Outcome 1:r1=1, 1:r2=0: Forbiden

(a) Code



(b) μhb graph

Figure 6.9: Modified `mp` [ABD+15] on a hypothetical GPU with no-allocate-on-write and no SWMR guarantee. In particular, the thread 1 fence does not enforce the load→load ordering, and so there is no cycle.

tween the cores and the memory system, and it is possible that fences do not take such buffering into account. This example demonstrates how CCICheck can be used to catch these subtle bugs in new architectures prior to their release.

6.6.2 Lazy Coherence

The second case study uses CCICheck to analyze CCI behavior in the recently proposed TSO-CC protocol [EN14]. TSO-CC is a scalable lazy coherence protocol for TSO architectures that uses private L1s and a shared L2 that doubles as a directory cache. TSO-CC does not track lines in shared state, allowing them instead to be lazily and automatically invalidated by the core, either due to natural eviction from the cache or after a certain fixed number of uses. The above results in low on-chip storage requirements for coherence but requires the pipeline to enforce orderings that it would not need to enforce in a more standard protocol. Scenarios like this in which a coherence protocol's design is tightly coupled with features of an MCM's implementation are great examples of the need for CCICheck.

Due to its lazy invalidation policy, the Single Writers/Multiple Readers policy is enforced logically rather than physically. In other words, although SWMR does not hold true in a strict physical sense, it does hold true in a more abstract sense: the cores may be considered non-synchronous, except at communication points, and SWMR holds according to this abstract timeline. To account for this non-synchrony, TSO-CC requires each core to "catch up" to a future point in logical time whenever sees a load miss or a fence by invalidating all shared lines in its private L1 cache (and hence any possibly out-of-date values) at that time. This ensures that when a core sees one value from a core other than itself, it is made aware of all previous values from all other cores as well.

We model TSO-CC in CCICheck by adjusting features in the baseline models as appropriate. A snippet of this code is shown in Figure 6.10. First, the Single Writ-

```
DefineMacro "L1ViCLFlushSharedLines":
  forall microops "i'",
    (NodeExists (i', L1ViCLCreate) \/ NodeExists (i', L1ViCLExpire)) =>
    ~SameMicroop i i' => SameCore i i' => (
      EdgeExists  ((i', L1ViCLExpire), (i, L1ViCLCreate), "Flush") \/
      (EdgeExists ((i, L1ViCLCreate), (i', L1ViCLDowngrade), "StillInModified") /\ IsAnyWrite i') \/
      EdgeExists  ((i, L1ViCLCreate), (i', L1ViCLCreate), "CreatedAfter")).

Axiom "L1ViCLs":
  forall microop "i",
    OnCore c i => IsAnyRead i =>
    (NodeExists (i, L1ViCLCreate) \/ NodeExists (i, L1ViCLExpire)) => (
      EdgeExists ((i, L1ViCLCreate), (i, L1ViCLExpire), "path") /\
      ExpandMacro L1ViCLSource /\ ExpandMacro L1ViCLFlushSharedLines).

Axiom "SWMR":
  forall microops "i1",
    (IsAnyWrite i1 \/ AccessType RMW i1) =>
    (NodeExists (i1, L1ViCLCreate) \/ NodeExists (i1, L1ViCLExpire)) =>
    forall microops "i2",
      ((NodeExists (i2, L1ViCLCreate) \/ NodeExists (i2, L1ViCLExpire)) =>
      ~SameMicroop i1 i2 => SameAddress i1 i2 =>
      (IsAnyWrite i2 \/ AccessType RMW i2) => (
        (EdgeExists ((i2, L1ViCLDowngrade), (i1, L1ViCLCreate), "swmr")) \/
        (EdgeExists ((i1, L1ViCLCreate   ), (i2, L1ViCLCreate), "swmr")))) /\
      ((NodeExists (i2, (0, L2ViCLCreate)) \/ NodeExists (i2, (0, L2ViCLExpire))) =>
        (~SameMicroop i1 i2) => SameAddress i1 i2 => (
          (EdgeExists ((i2, (0, L2ViCLExpire)), (i1,     L1ViCLCreate ), "swmr")) \/
          (EdgeExists ((i1,     L1ViCLCreate ), (i2, (0, L2ViCLCreate)), "swmr")))).
```

Figure 6.10: Relevant subset of the CCICheck axioms for TSO-CC.

ers/Multiple Readers (`SWMR`) axiom is weakened to ensure that only lines in modified state are affected; such lines are downgraded to shared, while existing shared lines (which are not tracked) are ignored. Likewise, the L1 ViCL sourcing axiom `L1ViCLs` is modified to state that for each ViCL matching the same address as a ViCL brought in from the L2 cache, either 1) the cache line was in shared state and must be flushed, 2) the cache line is in modified state (i.e., has not yet been downgraded) and may be kept as-is, and 3) the cache line is created after the flush occurs. Fences and atomic read-modify-write instructions behave similarly.

Figure 6.11 shows a μhb graph for the `mp` litmus test (Figure 6.2a) executing on a microarchitecture implementing TSO-CC. Since shared lines in the L1 caches are invalidated lazily, the L1 ViCL for the load of `x` need only be sourced before the store of `x` occurs as opposed to being invalidated before it. This relationship is identical to the case of the partially incoherent architecture covered in Section 6.6.1. Intuitively,

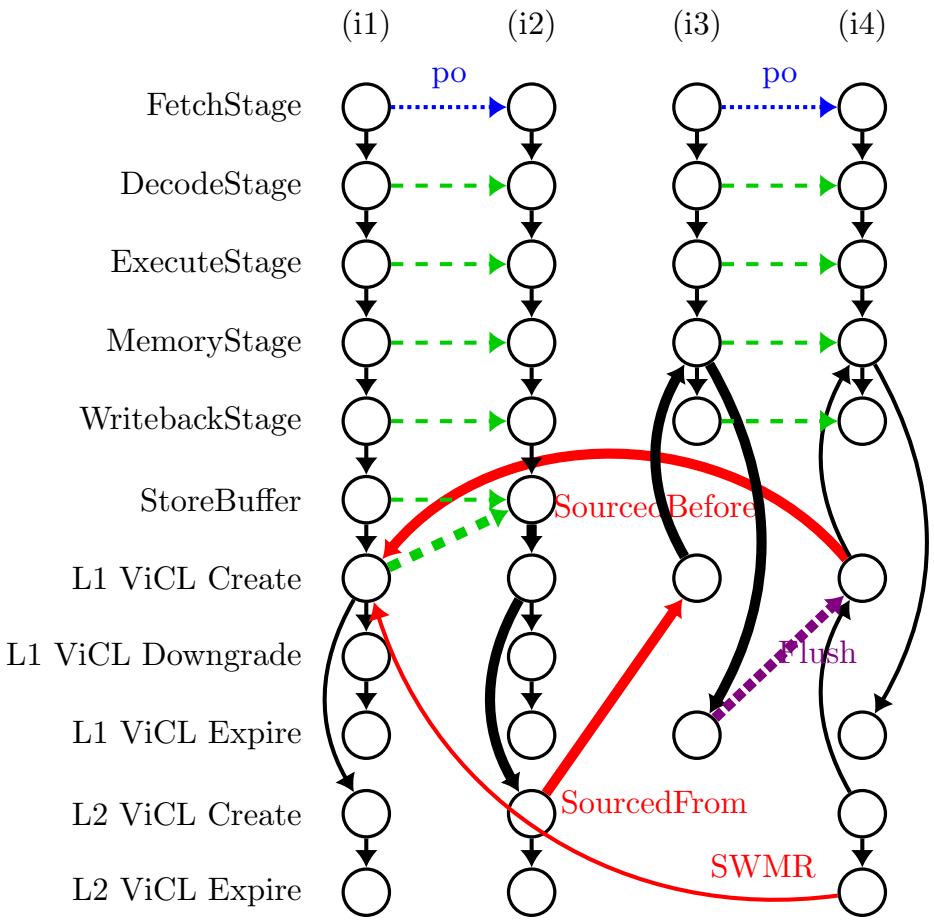


Figure 6.11: μhb graph for `mp` on TSO-CC [EN14]. Although the coherence protocol does not eagerly invalidate any sharers of `x` before allowing (i1) to perform, the necessary orderings are enforced by (i3) flushing the cache upon taking a cache miss.

in TSO-CC, if the load of `y` returns the value 1, it must have taken a cache miss at some point, allowing the new value of 1 to be fetched from core 0. Core 1 would have invalidated shared lines in its L1 cache at the time of this cache miss, and thus the ViCL for the load of `x` must be created after the ViCL for the load of `y` is created.

The litmus test specifies a forbidden outcome, so the verification goal is to identify a cycle in the μhb graph. If the TSO-CC implementation properly flushes the cache as specified, then the "Flush" edge will exist, creating a cycle in the graph and indicating that TSO-CC maintains TSO for this execution. If the "Flush" edge did not exist, the graph would be acyclic and the forbidden test outcome would be potentially observable, indicating that TSO-CC's flushing of shared L1 lines on a cache

miss is a critical part of how it maintains TSO. Not only does the ViCL abstraction allow for comprehensive CCI verification, but the automatically-enumerated μhb graphs give the designer considerable intuition about how and why correct behavior is preserved. Although the authors of TSO-CC performed testing via simulation, CCICheck is more comprehensive because it allows for exhaustive enumeration of possible ordering scenarios, where simulation may be limited to just a few different orderings or interleavings.

We also hope that CCICheck helps clarify statements such as the TSO-CC authors' claim that most coherence protocols are designed for sequential consistency [EN14]. As numerous widely-used MCMs such as TSO, ARM, and Power are considered coherent yet not sequentially consistent, it is easy to misinterpret such a claim. The coherence protocol cannot enforce the consistency model independently of pipeline components such as store buffers or of pipeline behaviors such as speculative load reordering. Our point of view is that both coherence protocols and pipelines *provide* certain (implementation-level) ordering properties, and that consistency models *require* certain ordering properties. A coherence protocol may be designed to provide the kinds of strong properties that a stricter consistency model (such as SC) may require, but it is nevertheless up to the system as a whole to ensure that consistency is properly enforced.

6.6.3 Window of Vulnerability/Peekaboo

This section demonstrates the use of CCICheck to verify the solution to the window of vulnerability (WoV) and Peekaboo problem previously introduced in Section 6.2.2. The solution required that loads and stores access already-invalidated data if and only if they were the oldest in program order at the time of the coherence request for the accessed line. The CCICheck model for the Peekaboo protocol therefore includes a Cache Request node and a Cache Line Invalidate node. The Cache Request node

```
DefineMacro "L1ViCLSource":
exists microop "i'", (
  SamePhysicalAddress i i' /\ ~SameMicroop i i' /\ SameData i i' /\
  EdgesExists ((i', ViCLCreate), (i , ViCLCreate    ), "src", "blue") /\
              ((i', ViCLCreate), (i', ViCLInvalidate), "DontSrcFromPeekabooViCL")] /\
  ~exists microop "i''",
    SamePhysicalAddress i i'' /\ IsAnyWrite i'' /\
    EdgesExist [((i' , ViCLCreate), (i'', ViCLCreate), "DVI");
                ((i'', ViCLCreate), (i  , ViCLCreate), "DVI")]).

DefineMacro "FindL1ViCLNormal":
exists microop "s", (
  SamePhysicalAddress s i /\ SameData s i /\ SameCore s i /\
  EdgesExist [((s, ViCLCreate ), (i, MemoryStage   ), "rf");
              ((i, MemoryStage), (s, ViCLInvalidate), "rf");
              ((s, ViCLCreate ), (s, ViCLInvalidate), "path")]).

DefineMacro "FindL1ViCLPeekaboo":
(~AccessType RMW i) /\
exists microop "s", (
  SamePhysicalAddress s i /\ SameData s i /\ SameCore s i /\
  EdgesExist [((s, ViCLInvalidate), (s, ViCLCreate ), "path");
              ((s, ViCLCreate ), (i, MemoryStage), "rf");
              ((i, MemoryStage), (s, ViCLExpire ), "rf")] /\
  forall microop "i'",
    ProgramOrder i' i => (
    (IsAnyRead  i' => EdgeExists ((i', MemoryStage), (s, ViCLRequest), "peekaboo")) /\
    (IsAnyWrite i' => EdgeExists ((i', ViCLCreate ), (s, ViCLRequest), "peekaboo")))).

DefineMacro "FindL1ViCL":
  ExpandMacro FindL1ViCLNormal \/ ExpandMacro FindL1ViCLPeekaboo.
```

Figure 6.12: Relevant subset of the CCICheck axioms for the Peekaboo scenario

represents the time at which a core makes a coherence request for a particular line, and the Cache Line Invalidate node represents the point at which an invalidation request is processed. Here, unlike the cases previously discussed, the invalidation of the ViCL is *not* the same as its expiration. This is because a load may return the invalidated data to one instruction in an effort to avoid livelock.

Figure 6.12 shows the key axioms used to model the Peekaboo scenario. Each load may choose one of two options: it may behave as a normal load (i.e., it may behave the same as in the baseline models), or it may be a Peekaboo load in which the cache line invalidation arrives before the data is used. In the latter case, we simply add a new set of μhb edges to represent the fact that the instruction must be the oldest in program order at the time of the coherence request: all previous instructions must have completed before the Cache Request for the instruction in question takes place.

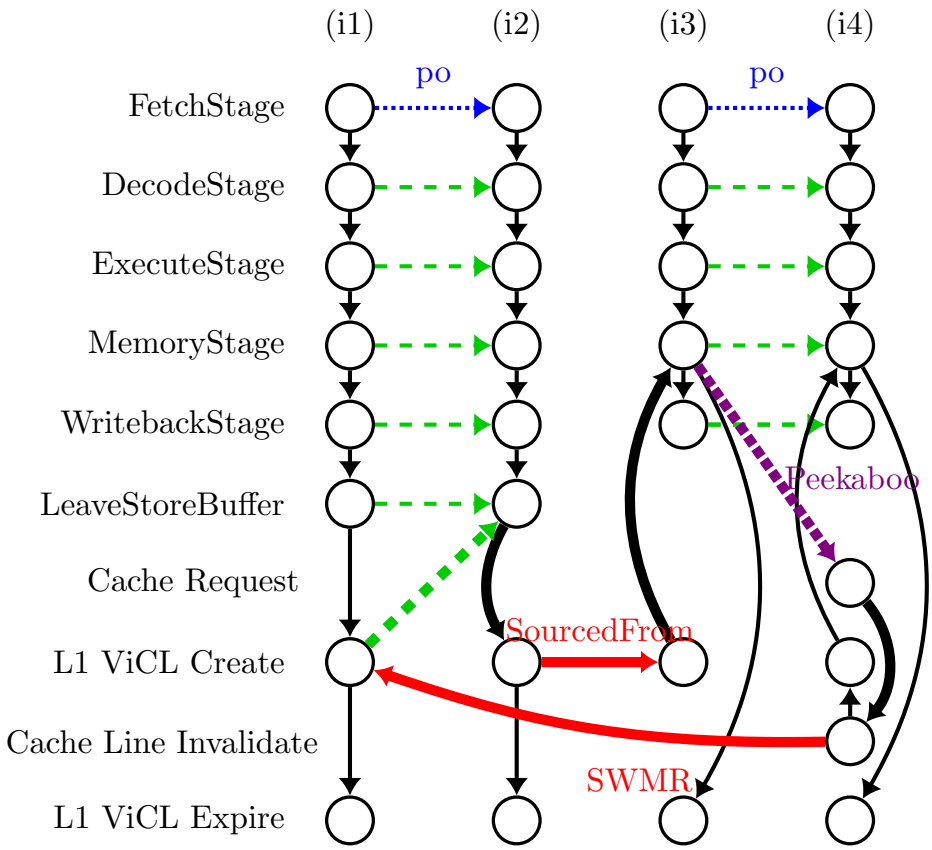


Figure 6.13: μhb graph for the solution to the Peekaboo coherence problem.

Figure 6.13 demonstrates a μhb graph for the Peekaboo solution technique, again for the `mp` litmus test. Since invalidated data is only returned to the core if the instruction it is returned to was the oldest in program order at the time of the coherence request, our μhb graph must be able to include happens-before edges that indicate that. In particular, consider the load (i4) which accessed invalidated data. The graph includes a μhb edge (in this case labeled "Peekaboo") from the memory stage of all preceding accesses on the same core (i.e., (i3)) to the Cache Request node of (i4). The presence of the Peekaboo μhb edge ensures that a cycle is created, thereby preventing the illegal outcome. Thus, CCICheck helps designers verify that as long as the implementation ensures that the requesting instruction was the oldest in program order at the time of the coherence request, this livelock avoidance approach abides by TSO consistency.

Finally, we note that (unsurprisingly) the act of formally specifying CCI behavior can sharpen a designer's understanding of system behavior. For example, the original English-language description of the problem [SHW11]—while very thorough and perhaps clear to many—required us to interpret where instructions preceding the Peekaboo load (in program order) must have progressed before performing the load. In creating the CCICheck model, we were able to use the tool to determine that a correct design would only allow the load to access invalidated data if all previous loads were *performed* and all previous stores had *reached the memory hierarchy and become visible to all cores* before the coherence request for the Peekaboo instruction was issued. By requiring formal and explicit documentation regarding such decisions, the CCICheck model becomes a more precise solution specification than any natural language description could be.

6.6.4 Performance Results

Figure 6.14 shows the runtimes of CCICheck across the x86-TSO litmus test suite. As in Section 5.6, there can be significant variation in the time it takes CCICheck to run a litmus test across tests and across microarchitectures. This is to be expected as litmus tests have varying numbers of instructions and varying potential for ViCL pairing and thus different ranges of possibilities. However, even though the graphs and the models are larger than they were in the previous chapter, and even though the ViCL abstraction adds complexity to the constraints that must be satisfied, the runtimes clearly still remain very tractable.

Just as in the previous chapter, the published CCICheck solver runtimes were originally much longer [MLPM15], with some tests taking as long as hours and large amounts of memory. The development of the solver of Section 5.4.2 provided a much-needed boost in performance while maintaining (and through the use of the DSL, even improving) the generality of the approach.

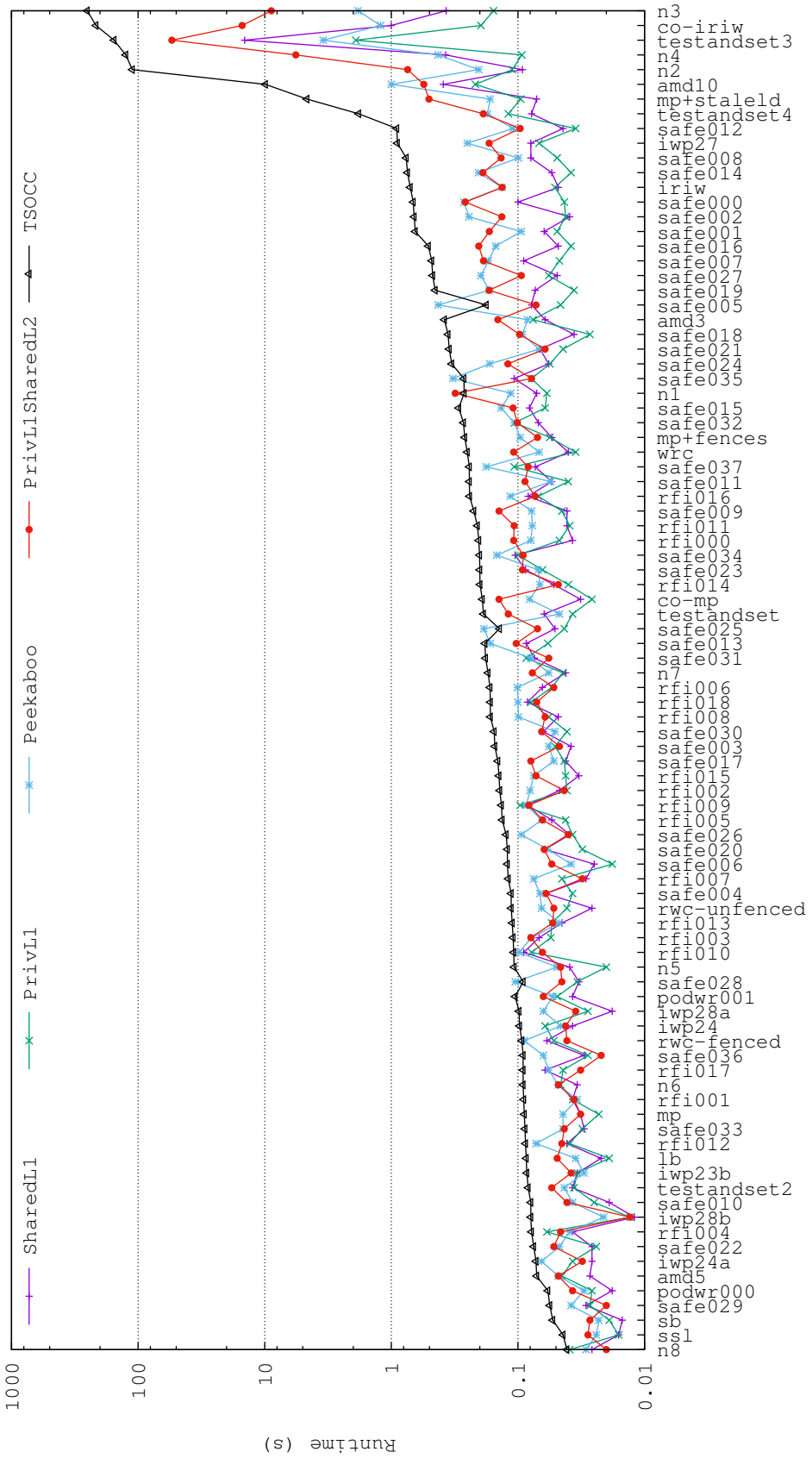


Figure 6.14: Runtimes of the x86-TSO litmus tests across the various processors we modeled.

6.7 Related Work

CCI Definition and Verification. It is a compelling notion that coherence and consistency should be decoupled [Mar05]. Because of this, and because verification of each individually is more tractable than combined verification, much of the prior work discussed in this section focuses primarily on one or the other. Unfortunately, since most implementations of coherence and consistency become interwoven for performance or design reasons [ABD+15, ADC11, CKS+11, EN14, KSB95, RK12, SHW11], such a separation is insufficient, and verification of the CCI is an under-researched necessity. CCICheck's μhb graphs, ViCL abstractions, and axiomatic treatment of coherence protocol behaviors allow it to fill the CCI verification gap.

Computational Complexity. Alur et al. showed that verifying that a protocol correctly implements sequential consistency can in the worst case be formally undecidable [AMP96]. However, follow-up work later showed that verification of SC given certain realistic assumptions about loads not being able to receive data from future stores is in some cases decidable [BCH03, CH03]. Verifying memory coherence, as a decision problem, is NP-complete if coherence is considered equivalent to verifying sequential consistency for programs with only one variable [CLS03]. Furthermore, verifying sequential consistency remains NP-hard even if coherence is assumed to already hold true [CLS03].

In spite of the theoretical intractability of many consistency model verification problems, researchers have proposed numerous static and dynamic analysis tools which have been able to verify practically-relevant cases [AMT14, CMP08, HVML04, MS05, SSA+11]. Furthermore, many authors conjecture that consistency model behavior can be captured through the understanding of certain well-defined patterns, and these patterns can be directly encoded as litmus tests and/or sets of constraints in a model [AMT14]. These litmus tests tend to be small, consisting of no more

Ref.	Protocols	Coherence	Liveness	Deadlock Freedom	Safety Properties
[CGH+93]	Futurebus+	—	—	—	✓
[PSCH98]	SGI Origin 2000-like	SC	—	—	—
[McM01]	FLASH [KOH+94]	SC per loc.	✓	—	—
[CMP04]	German [PRZ01], FLASH	—	—	—	✓
[SSH+13]	Chained Cache Coh.	MOESI, DVI	—	✓	—
[STM14]	FLASH, German	—	—	system-wide	—

Table 6.2: A brief sampling of coherence protocol verification approaches

than dozen instructions or so. As in this chapter and the previous chapter, the small constant factors keep our analysis tractable in practice.

Coherence Verification. The field of cache coherence protocol verification has also seen significant attention from architecture researchers, from formal methods researchers, and from industry. Early work in this realm explored the verification of the FutureBus+ protocol using temporal logic model checking [CGH+93]. In modern research, formal modeling languages such as Murphi or TLA+ are often used to specify and to verify properties of many coherence protocols [DDHY92, Lam02, McM01, SSH+13, STM14]. Some of these projects directly verify coherence (for some definition of coherence, see Section 2.1.3). Some verify other properties: for example, freedom from deadlock, a liveness guarantee, or various safety properties concerning certain invariants that should always hold true or that should never hold true in any state. Just as the definition of coherence varies across different studies, the properties being verified vary widely. Table 6.2 gives a very brief sampling of how approaches differ in this regard. Among the protocols verified, the German protocol is notable for being designed to serve as a challenge for formal verification methods [CMP04]. More recently, Zhang et al. [ZBES14, ZLS10] have proposed techniques for designing cache coherence protocols in ways that make them inherently more amenable to formal verification. As stated earlier, PipeCheck interprets the properties verified by the above research as axioms which it then uses to verify the correctness of a consistency model implementation built on top of such protocols.

Vijayaraghavan et al. prove the correctness of a parameterized processor design written in BlueSpec [Inc04, VCAD15]. They demonstrate that the processor as a whole, including multiple speculative out-of-order cores and a directory-based coherence protocol with an arbitrary number of nodes and levels, correctly implements sequential consistency. Their approach, however, used manual proofs written in Coq [The04], and it does not yet support verification of memory models weaker than sequential consistency PipeCheck supports automated verification of a broader range of weak memory models.

6.8 Chapter Summary

This chapter demonstrated that the μhb analysis techniques of the previous chapter can be readily extended to model the consistency-related features of a specific coherence protocol. More broadly, it demonstrated how coherence and consistency are inherently intertwined at a microarchitecture level, and that naive extensions of architecture-level relationships such as `rf`, `ws`, and `fr` do not always correspond to reality. To address this, we introduced the notion of a value-in-cache-lifetime, or ViCL, which allowed us to replace the no-longer-applicable edges with cache-focused axioms such as the Single Writers/Multiple Readers (SWMR) axiom. Using this strategy, CCICheck builds off of previous work in coherence protocol verification [CGH+93, McM01, SSH+13, ZBES14, ZLS10] rather than competing with or reproducing it. This breeds confidence in the potential for future extensions to PipeCheck and CCICheck to be able to capture the behavior of particular networks-on-chip, address translation subsystems, and so on, thereby improving the tractability of whole-chip verification in the real world.

Chapter 7

Conclusion and Future Directions

This section describes ongoing and future work, and it then concludes the thesis.

7.1 Future Directions

There are many important and exciting possible ways in which the work in this thesis can be further extended. Some of these possibilities are considered below.

7.1.1 Further Extensions of PipeCheck

One exciting avenue for future work to extend PipeCheck and CCICheck to analyze broader notions of correctness that consistency models traditionally address.

One extension to PipeCheck which is ongoing as this thesis is being written is the extension of PipeCheck to verify the intersection of memory consistency models with address translation. Romanescu et al. demonstrated that consistency models for virtual addresses may differ from consistency models for physical addresses [RLS10]. They also demonstrated the need for consistency verification methods to take into account address remapping and permission changing functions such as `mmap` and `mprotect`. Along with some collaborators, I am currently developing ATCheck to

explicitly model both virtual and physical addresses, memory mapping/remapping functions, OS effects related to TLB maintenance, and other related mechanisms. I believe that ATCheck will be able to help resolve bugs (which are unfortunately all too common) in the address translation subsystem, just as PipeCheck and CCICheck were able to do for the pipeline and cache coherence protocol, respectively.

More broadly, I believe that the PipeCheck DSL and software have the potential to find widespread adoption in the architecture community. The PipeCheck DSL is flexible enough to describe orderings at many different levels of abstraction, thereby allowing users to describe their ideas at the level of abstraction that makes the most sense for the given scenario and/or that they are most comfortable using. The software is fast enough to provide quick or even nearly-instant feedback to the user, thereby providing a lower barrier to entry for architects wishing to validate their work. My hope is that one day, PipeCheck (or an extension thereof) will become a standard for demonstrating the MCM correctness of any new architectural design, much as how verification of a coherence protocol using, e.g., Murphi or TLA+ has become a de facto requirement when proposing new coherence protocol designs [DDHY92, Lam02].

Lastly, I believe that μhb graphs and the axiom-based approach of the PipeCheck DSL will be able to serve as a valuable teaching tool for students learning how memory consistency models are defined and implemented. A common criticism of memory models in general is that they are highly confusing and opaque, and this often makes it difficult for students to fully grasp the details of memory model analysis techniques or even to fully appreciate the need for such techniques in the first place. The PipeCheck approach provides two key advantages in this regard. First, it decomposes the problem of specifying memory models into smaller axioms which are often finer-grained and more concrete (in the sense of representing some specific microarchitectural behavior), making them easier to comprehend. Second, it supplements

dense mathematical formalisms with easily-visualizable μhb graphs, thereby hopefully increasing the accessibility of the topic to a broader audience.

7.1.2 Formal Correctness Proofs

Although the PipeCheck analysis framework has been implemented within Coq, it has not yet taken advantage of the Coq's ability to build formal proofs of correctness. The first challenge, as stated in Section 5.4.1, is that there are many cases in which a proper architecture-level consistency model specification simply does not exist. Nevertheless, for cases in which a specification does exist, it should be possible to derive fully formal correctness specifications using the approach laid out in Section 5.4.6. It may well also be possible to leverage the ability of PipeCheck to perform automated exploration of the search space in order to generate proofs or proof components entirely automatically.

Likewise, our verification approach is only as sound as our microarchitecture models are with respect to the actual underlying microarchitectures. A follow up project that would be very beneficial would be to rigorously compare the behavior of a design at the register transfer level (RTL) to the axioms stated in our microarchitecture models. With both of the above, it would be possible to build a complete formal proof correctness from the software level all the way down to the transistor level.

Lastly, one can explore how ArMOR might be formalized more rigorously as well. This could help lend increased reliability to case studies such as the dynamic translation shims of Chapter 4. It could also help in the process of integrating ArMOR and PipeCheck to form a complete and unified framework for analyzing consistency models at the hardware level and below.

7.2 Thesis Conclusions

Exciting times lie ahead for the field of computer architecture. Dennard scaling has seemingly faltered, and the continued transistor density doubling predicted by Moore's law will eventually reach its limit. Even Intel, currently the world's leading semiconductor manufacturer, announced that its 10nm silicon fabrication process would be delayed by a year [HS15]. While it is impossible to predict the future, it is not inconceivable to think that the future of computer architecture may soon look dramatically different than the silicon-based world of today. From a research perspective, this future presents a fascinating new world just waiting to be explored.

It is fairly clear that architectures of the near future will become increasingly specialized and heterogeneous. Accelerators are already ubiquitous; GPUs are widespread, and mobile systems-on-chip in particular already make significant use of specialized compute units in order to meet their performance and power targets. Furthermore, systems are increasingly interconnected, both directly and indirectly through the Internet. In this setting, communication between components and between devices (via shared memory or otherwise) will be of paramount importance for performance and for correctness. Therefore, techniques like PipeCheck become all the more central.

If and when new materials and device technologies supplant the traditional silicon-based paradigm, they could very well introduce questions analogous to the questions addressed by existing memory consistency models. Many exciting new memory technologies are being studied and implemented in modern systems; examples such as flash and 3D-stacked DRAM have already gained commercial acceptance. They also raise new questions. For example, many of these new memories are persistent—they do not lose their state when power is removed. As a consequence, researchers have also begun to consider the correctness implications of memory "persistency", the set of rules for enforcing orderings between the points when memory accesses become

persistent [PCW14]. For these kinds of complex designs, achieving programmability without sacrificing performance, power efficiency, and correctness will require robust and flexible frameworks for specifying and analyzing memory consistency and any other form of correctness.

This thesis makes various contributions to the state of the art of computer architecture and memory consistency model analysis techniques. Building off of a growing body of research demonstrating the need for rigor when specifying and verifying memory consistency models, this thesis presents new techniques for extending rigorous memory model analysis down to the microarchitecture layer. ArMOR brings much-needed clarity to the widely-used yet fundamentally-flawed informal specifications used by many industry products and academic projects. PipeCheck then derives the first formal microarchitecture-level memory models and verifies their behavior against a given architecture-level specification.

Overall, this thesis makes the following contributions:

- Chapter 3 presents a novel unifying framework for specifying and comparing memory models. The MOST syntax defines a single precise, self-contained, and architecture-independent framework that can be used to define the memory models of many widely-used architectures. The ArMOR framework then allows for the systematic comparison, analysis, and manipulation of MOSTs, thereby allowing MOST-based consistency model analysis to be embedded within compilers, dynamic binary translators, or other software or hardware components. ArMOR allows features such as preserved program order and fences to be directly compared to each other, even when they originate from different models. This ensures that such analyses are less prone to the kinds of subtle bugs that often appear in memory model analyses. The flexibility of ArMOR also allows memory model analysis techniques to be extended to heterogeneous systems with components implementing more than one type of memory model. This

flexibility will become invaluable as architectures become increasingly heterogeneous and, following the progression of accelerators such as GPUs, as components are increasingly expected to share memory flexibly and at fine granularity.

- Chapter 4 provides a concrete example of how the techniques developed in this thesis can be used to enable new technologies that simply could not exist prior to this thesis. The example used in this chapter, the derivation of dynamic binary translation shims which account for differences in memory consistency models between the source and target architectures, fills a crucial gap that previous dynamic binary translators and emulators were unable to solve [DVT12, Qem15, VT14]. MOSTs and the ArMOR framework provide an elegant solution to these problems by presenting the first general-purpose algorithm for translating between any two memory models provided as inputs. More broadly, this chapter presents an example of how the algorithms enabled by ArMOR can easily scale to the needs of forward-looking technologies.

- Chapters 5 and 6 present the first general-purpose framework for rigorously specifying and verifying memory consistency model behavior at the microarchitecture-level. While the past decade has seen significant progress made to improve the specifications of software models such as C++ and Java and of hardware models such as x86-TSO, Power, and ARM, it has seen little similar effort to verify that architecture-level models were being implemented correctly. PipeCheck and CCICheck present allow formal analysis techniques to extend down to the implementation level, thereby enabling proofs of correct behavior to extend from software all the way down to a particular microarchitecture. They also allow microarchitectures which do not yet have formal (or even informal) memory models (e.g., most accelerators) or which rely on microarchitecture-specific just-in-time (JIT) compilation paradigms

(e.g., GPUs) to nevertheless be rigorously analyzed. The PipeCheck approach derives architecture-level correctness from a set of localized implementation-level axioms. The approach also enables a new, more tractable path towards extending verifiability all the way down to the transistor level.

Although memory consistency models have been studied since the 1970s, they remain an important and active area of research. Decades of effort have shown that memory models are more than just an engineering problem. Most attempts to define sound models which enable high-performance implementations and which are programmable by non-"experts" [BA08] tend to start their life flawed and only improve after years of iteration [AMT14, AŠ07, BA08, MPA05, SSA+11]. Many popular models are still iterating even today [ABD+15, BMO+12, SMO+12, VBC+15]. Given the trend toward increased heterogeneity and specialization, problems of cross-component communication and synchronization are likely only to increase in importance.

This thesis makes a number of important contributions to memory consistency model research from a microarchitect's point of view, and it does so in a way that balances the theoretical need for formalism and rigor with the practical need to be able to analyze real systems. The analysis techniques introduced here can help eliminate some of the consistency-related hardware and software bugs that continue to appear in real widely-used systems. They can also help shed much-needed light on why memory models are inherently complicated yet fundamentally important, how memory model enforcement mechanisms are implemented, and how all of the above will need to be adapted to the architectures of the future.

Appendix A

Gallery of MOST-based Definitions for Well-Known Architectures

The following appendix presents a gallery of MOSTs for well-known architectures. Its purpose is twofold. First, it demonstrates how MOSTs can be used to describe the MORs of many highly-relevant hardware memory models. Second, it provides a demonstration of how ordering enforcement mechanisms differ across these different architectures. All of the MOSTs in this appendix have been refined to distinguish same vs. different address relationships and to account for cumulativity, even though many of the MOSTs may otherwise be defined in simpler forms.

We model the following architectures:

- Sequential Consistency (SC) [Lam79].
- SPARC hierarchy: total store ordering (TSO), partial store ordering (PSO), and relaxed memory ordering (RMO) [OSS09a, SPA94b]. Note that although the word "relaxed" is often used as a generic term, in the context of SPARC RMO it has a well-defined meaning.

- POWER and ARM [ARM13, AMT14, IBM13, MHMS+12, SSA+11]. Note that our MOSTs treat loads as single events, while recent formalizations treat them as consisting of multiple events. This causes our MOST-based Power model to be strictly stronger than the standard Power model, with discrepancies in 394 (5%) out of 7536 litmus tests from an existing suite of litmus tests for the Power architecture. The ARM model is likewise stronger than standard ARM models. This could be corrected by adding strength levels (Section 3.3.1) which distinguish between the multiple load events as well.

- NVIDIA PTX [ABD+15, NVI13b]. Note that 1) this model is not yet as well formalized as the other models described above, and 2) this model does *not* provide a way to enforce cumulativity, and it therefore cannot be made to behave in a sequentially consistent manner, in contrast with essentially all other general-purpose MCMs.

The code which automatically generated these shims (and this appendix) is available online [Lus15]: `https://github.com/daniellustig/armor`.

A.1 Sequential Consistency (SC)

	PPO					
	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$

Figure A.1: MOSTs for MCM Sequential Consistency (SC)

A.2 SPARC® Total Store Ordering (TSO)

PPO

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	—	✓$_L$	✓$_S$	✓$_M$	✓$_M$	✓$_S$

mfence

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$

Figure A.2: MOSTs for MCM SPARC® Total Store Ordering (TSO)

A.3 SPARC® Partial Store Ordering (PSO)

PPO

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	—	✓$_L$	✓$_S$	—	✓$_M$	✓$_S$

mfence

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$

Figure A.3: MOSTs for MCM SPARC® Partial Store Ordering (PSO)

A.4 SPARC® Relaxed Memory Ordering (RMO)

PPO

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	—	✓$_S$	✓$_S$	—	✓$_S$	✓$_S$
PO ST	—	✓$_L$	✓$_S$	—	✓$_M$	✓$_S$

mfence

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$

Figure A.4: MOSTs for MCM SPARC® Relaxed Memory Ordering (RMO)

A.5 Power Architecture®

PPO

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	—	—	—	—	—	—
AC ST	—	—	—	—	—	—
PO LD	—	✓$_S$	—	—	✓$_S$	—
PO ST	—	✓$_L$	—	—	✓$_N$	—

dep

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	—	—	—	—	—	—
AC ST	—	—	—	—	—	—
PO LD	✓$_S$	✓$_S$	—	✓$_S$	✓$_S$	—
PO ST	—	✓$_L$	—	—	✓$_N$	—

lwsync

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	—	—	—	✓$_N$	✓$_N$	✓$_N$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	—	✓$_L$	—	✓$_N$	✓$_N$	✓$_N$

sync

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$

Figure A.5: MOSTs for MCM Power Architecture®

A.6 ARM® Architecture

PPO

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	—	—	—	—	—	—
AC ST	—	—	—	—	—	—
PO LD	—	✓$_S$	—	—	✓$_S$	—
PO ST	—	✓$_L$	—	—	✓$_N$	—

dep

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	—	—	—	—	—	—
AC ST	—	—	—	—	—	—
PO LD	✓$_S$	✓$_S$	—	✓$_S$	✓$_S$	—
PO ST	—	✓$_L$	—	—	✓$_N$	—

dmb

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$	✓$_S$

Figure A.6: MOSTs for MCM ARM® Architecture

A.7 NVIDIA® PTX

PPO

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	—	—	—	—	—	—
AC ST	—	—	—	—	—	—
PO LD	—	✓$_S$	—	—	✓$_S$	—
PO ST	—	✓$_L$	—	—	✓$_N$	—

membar

	PO LD/DA	PO LD/SA	BC LD	PO ST/DA	PO ST/SA	BC ST
AC LD	—	—	—	—	—	—
AC ST	—	—	—	—	—	—
PO LD	✓$_S$	✓$_S$	—	✓$_S$	✓$_S$	—
PO ST	✓$_S$	✓$_S$	—	✓$_S$	✓$_S$	—

Figure A.7: MOSTs for MCM NVIDIA® PTX

Appendix B

Gallery of Shim FSMs

The following appendix presents a gallery of MOSTs and shim FSMs used in the evaluation of Chapter 4. The goal of the appendix is to demonstrate 1) the breadth of applicability of ArMOR, and 2) to demonstrate that shims are generally no more than a few states

In the implementation of Section 4.4.2, addresses may not all be resolved by the time instructions reach the issue queue. The analysis in this appendix ignores the distinction between same- vs. different-address relationships when deriving shim designs. Instead, it chooses the more conservative option in each case.

The following architectures are considered in this appendix:

- The models from Appendix A: SC, SPARC TSO, SPARC PSO, SPARC RMO, Power, and ARM, except that the shims do not distinguish same vs. different address relationships.
- Partial load ordering (PLO), named by analogy to SPARC PSO, and load-store ordering (LSO), a slightly more permissive variant. These provide more variety within the SPARC hierarchy [MHAM11, SPA94b].
- RMO16: like SPARC RMO, but with a finer-grained set of fences.

- PowerA: a multi-copy atomic variant of Power, added as a comparison point and to demonstrate how shims targeting this variant can often be more intelligent than shims targeting Power, as the lack of multi-copy atomicity in the latter introduces unrecoverable overheads (Section 4.6)

Lastly, the caveat from Appendix A about Power and ARM being stronger than the standard definitions remains true. This turns out not to affect the results of this appendix. When Power/ARM are upstream models, it is conservative to assume a stricter-than-necessary specification of the model, and so this does not pose a problem. When Power or ARM serves as the downstream model, all of the shims already require `sync` or `dmb`, respectively, to be inserted before every instruction emitted downstream, so the discrepancy turns out not to pose a problem.

The code which automatically generated these shims (and this appendix) is available online [Lus15]: `https://github.com/daniellustig/armor`.

	TSO	PLO	PSO	LSO	RMO	RMO16	POWERA	POWER	ARM
SC	2	1	2	1	1	2	2	1	1
TSO	-	2	2	4	3	5	4	1	1
PLO	-	-	2	2	2	4	3	1	1
PSO	-	2	-	2	3	3	2	1	1
LSO	-	-	-	-	2	2	2	1	1
RMO	-	-	-	-	-	1	1	1	1
POWERA	-	-	-	-	1	1	-	1	1
POWER	-	-	-	-	-	-	-	-	1
ARM	-	-	-	-	-	-	-	1	-

Figure B.0.a: FSM node count summary.

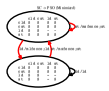
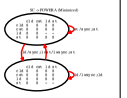
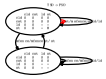
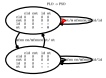
	TSO	PLO	PSO	LSO	RMO	RMO16	POWERA	POWER	ARM
SC									
TSO	-								
PLO	-	-							
PSO	-		-						
LSO	-	-	-	-					
RMO	-	-	-	-	-				
POWERA	-	-	-	-			-		
POWER	-	-	-	-	-	-	-	-	
ARM	-	-	-	-	-	-	-		-

Figure B.0.b: FSM summary.

B.1 Upstream MCMs

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$

Figure B.1.a: MOSTs for Upstream SC

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	—	✓$_S$	✓$_M$	✓$_S$

mfence

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$

Figure B.1.b: MOSTs for Upstream TSO

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	—	✓$_S$	✓$_S$	✓$_S$
PO ST	—	✓$_S$	✓$_M$	✓$_S$

mfence

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$

Figure B.1.c: MOSTs for Upstream PLO

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	—	✓$_S$	—	✓$_S$

mfence

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$

Figure B.1.d: MOSTs for Upstream PSO

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	—	✓$_S$	✓$_S$	✓$_S$
PO ST	—	✓$_S$	—	✓$_S$

mfence

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$

Figure B.1.e: MOSTs for Upstream LSO

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	—	✓$_S$	—	✓$_S$
PO ST	—	✓$_S$	—	✓$_S$

mfence

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$

Figure B.1.f: MOSTs for Upstream RMO

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	—	$\checkmark_S$
PO ST	—	$\checkmark_S$	—	$\checkmark_S$

lwsync

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_N$	$\checkmark_S$

sync

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

Figure B.1.g: MOSTs for Upstream POWERA

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	—	—	—	—

lwsync

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	—	—	$\checkmark_N$	$\checkmark_N$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	—	$\checkmark_N$	$\checkmark_N$

sync

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

Figure B.1.h: MOSTs for Upstream POWER

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	—	—	—	—

dmb

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

Figure B.1.i: MOSTs for Upstream ARM

B.2 Downstream MCMs

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_M$	$\checkmark_S$

msfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

mlfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_M$	$\checkmark_S$

mfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

Figure B.2.a: MOSTs for Downstream TSO

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_M$	$\checkmark_S$

msfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

mlfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_M$	$\checkmark_S$

mfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

Figure B.2.b: MOSTs for Downstream PLO

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	—	$\checkmark_S$

msfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

mlfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	—	$\checkmark_S$

mfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

Figure B.2.c: MOSTs for Downstream PSO

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	—	$\checkmark_S$

msfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

mlfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	—	$\checkmark_S$

mfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

Figure B.2.d: MOSTs for Downstream LSO

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	—	$\checkmark_S$
PO ST	—	$\checkmark_S$	—	$\checkmark_S$

msfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

mlfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	—	$\checkmark_S$

mfence

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

Figure B.2.e: MOSTs for Downstream RMO

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	—	$\checkmark_S$
PO ST	—	$\checkmark_S$	—	$\checkmark_S$

fence_LL_SS

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	—	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

fence_LL_LS_SS

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

fence_SL_SS

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	—	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

fence_SS

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	—	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

fence_LS_SL

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	—	$\checkmark_S$

fence_LL_SL_SS

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	—	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

fence_LL_LS_SL_SS

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

fence_LL

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	—	$\checkmark_S$
PO ST	—	$\checkmark_S$	—	$\checkmark_S$

fence_LS_SL_SS

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

fence_LS

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	—	$\checkmark_S$

fence_LS_SS

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

fence_SL

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	—	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	—	$\checkmark_S$

fence_LL_LS

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	—	$\checkmark_S$

fence_LL_LS_SL

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	—	$\checkmark_S$

fence_LL_SL

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	—	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	—	$\checkmark_S$

Figure B.2.f: MOSTs for Downstream RMO16

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	—	$\checkmark_S$
PO ST	—	$\checkmark_S$	—	$\checkmark_S$

lwsync

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_N$	$\checkmark_S$

sync

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

Figure B.2.g: MOSTs for Downstream POWERA

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	—	—	—	—

lwsync

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	—	—	$\checkmark_N$	$\checkmark_N$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	—	$\checkmark_N$	$\checkmark_N$

sync

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

Figure B.2.h: MOSTs for Downstream POWER

PPO

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	—	—	—	—

dmb

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$

Figure B.2.i: MOSTs for Downstream ARM

B.3 SC Upstream, TSO Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	$\checkmark_S$	—	$\checkmark_S$	—

Figure B.3.a: PPO of (Upstream SC - Downstream TSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	0
0	SSSS SSSS SS-- SS--	st	st	1
1	SSSS SSSS SS-- SSS-	ld	mlfence; ld	0
1	SSSS SSSS SS-- SSS-	st	st	1

Figure B.3.b: FSM Transition Table

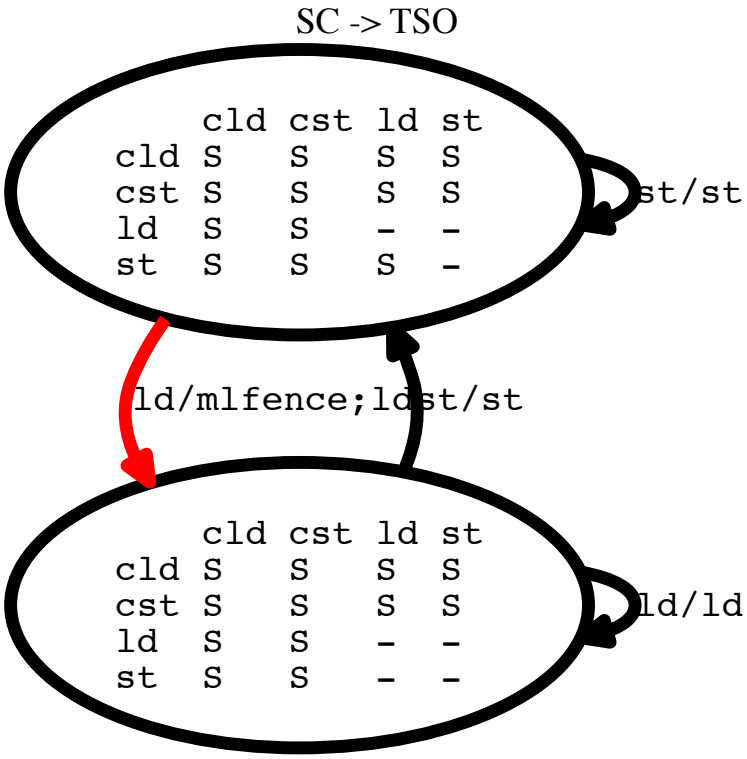


Figure B.3.c: FSM

B.4 SC Upstream, PLO Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	$\checkmark_S$	—	—	—
PO ST	$\checkmark_S$	—	$\checkmark_S$	—

Figure B.4.a: PPO of (Upstream SC - Downstream PLO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS*- SS*-	ld	mlfence; ld	0
0	SSSS SSSS SS*- SS*-	st	st	0

Figure B.4.b: FSM Transition Table

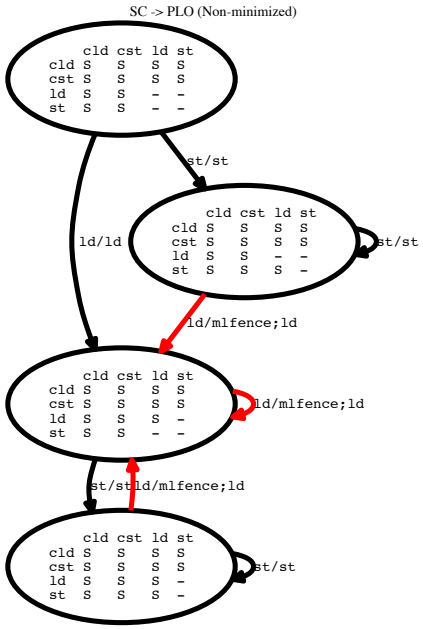


Figure B.4.c: FSM (Pre-minimization)

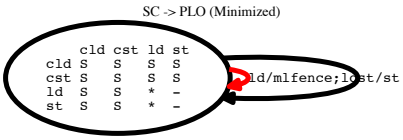


Figure B.4.d: FSM

B.5 SC Upstream, PSO Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	$\checkmark_S$	—	$\checkmark_S$	—

Figure B.5.a: PPO of (Upstream SC - Downstream PSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SSSS	ld	mlfence; ld	1
0	SSSS SSSS SS-- SSSS	st	msfence; st	0
1	SSSS SSSS SS-- SS-S	ld	ld	1
1	SSSS SSSS SS-- SS-S	st	msfence; st	0

Figure B.5.b: FSM Transition Table

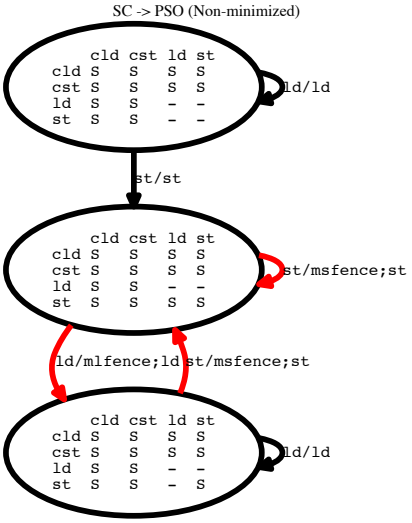


Figure B.5.c: FSM (Pre-minimization)

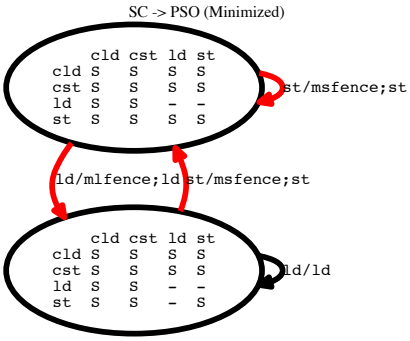


Figure B.5.d: FSM

B.6 SC Upstream, LSO Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	$\checkmark_S$	—	—	—
PO ST	$\checkmark_S$	—	$\checkmark_S$	—

Figure B.6.a: PPO of (Upstream SC - Downstream LSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS*- SS*S	ld	mlfence; ld	0
0	SSSS SSSS SS*- SS*S	st	msfence; st	0

Figure B.6.b: FSM Transition Table

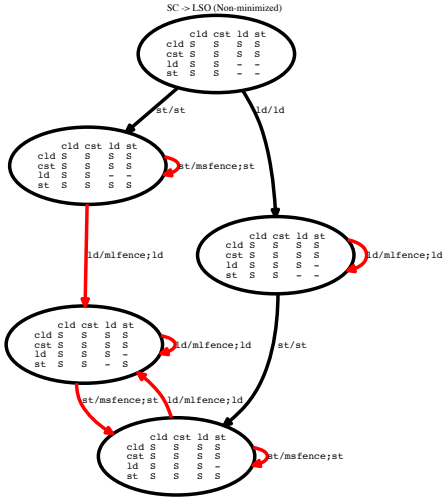


Figure B.6.c: FSM (Pre-minimization)

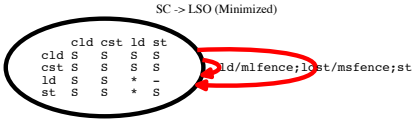


Figure B.6.d: FSM

B.7 SC Upstream, RMO Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	$\checkmark_S$	—	$\checkmark_S$	—
PO ST	$\checkmark_S$	—	$\checkmark_S$	—

Figure B.7.a: PPO of (Upstream SC - Downstream RMO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS** SS**	ld	mlfence; ld	0
0	SSSS SSSS SS** SS**	st	msfence; st	0

Figure B.7.b: FSM Transition Table

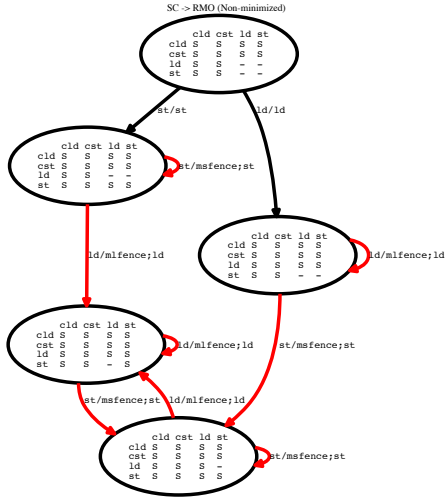


Figure B.7.c: FSM (Pre-minimization)

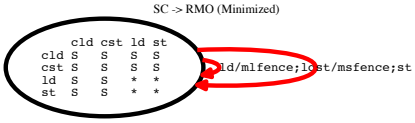


Figure B.7.d: FSM

B.8 SC Upstream, RMO16 Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	✓$_S$	—	✓$_S$	—
PO ST	✓$_S$	—	✓$_S$	—

Figure B.8.a: PPO of (Upstream SC - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SSS- SSSS	ld	fence_LL_SL; ld	1
0	SSSS SSSS SSS- SSSS	st	fence_SS; st	0
1	SSSS SSSS SSSS SS-S	ld	fence_LL; ld	1
1	SSSS SSSS SSSS SS-S	st	fence_LS_SS; st	0

Figure B.8.b: FSM Transition Table

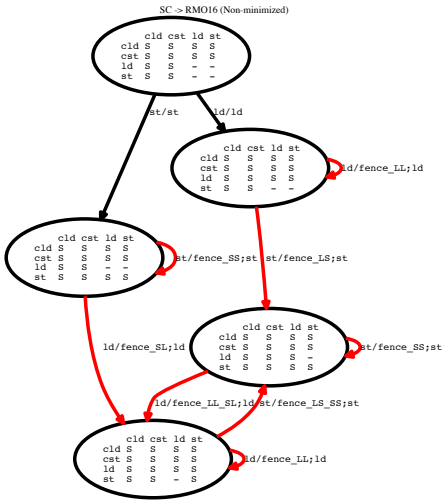


Figure B.8.c: FSM (Pre-minimization)

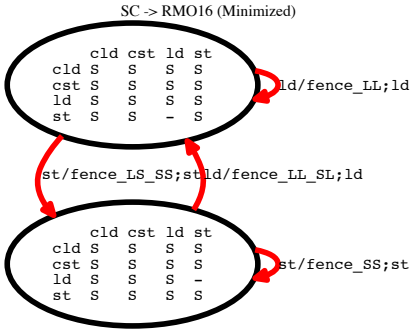


Figure B.8.d: FSM

B.9 SC Upstream, POWERA Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	✓$_S$	—	✓$_S$	—
PO ST	✓$_S$	—	✓$_S$	—

Figure B.9.a: PPO of (Upstream SC - Downstream POWERA)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SSSS	ld	sync; ld	1
0	SSSS SSSS SS-- SSSS	st	sync; st	0
1	SSSS SSSS SSSS SS--	ld	lwsync; ld	1
1	SSSS SSSS SSSS SS--	st	lwsync; st	0

Figure B.9.b: FSM Transition Table

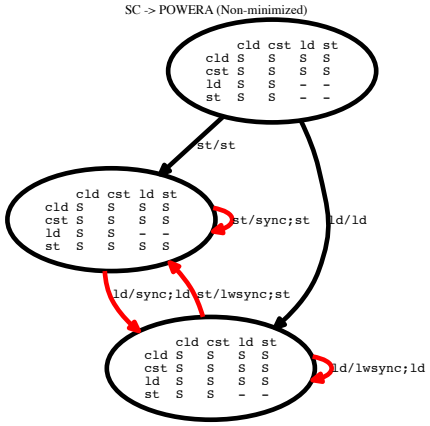


Figure B.9.c: FSM (Pre-minimization)

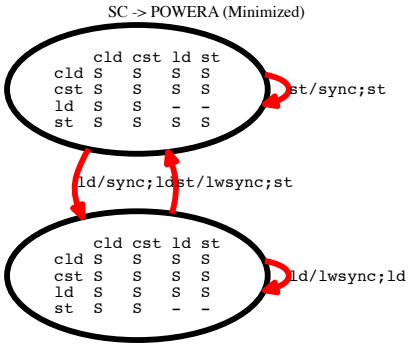


Figure B.9.d: FSM

B.10 SC Upstream, POWER Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	✓S	✓S	✓S	✓S
AC ST	✓S	✓S	✓S	✓S
PO LD	✓S	✓S	✓S	✓S
PO ST	✓S	✓S	✓S	✓S

Figure B.10.a: PPO of (Upstream SC - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS** SS**	ld	sync; ld	0
0	SSSS SSSS SS** SS**	st	sync; st	0

Figure B.10.b: FSM Transition Table

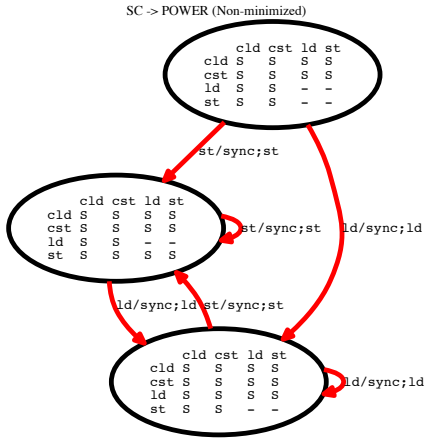


Figure B.10.c: FSM (Pre-minimization)

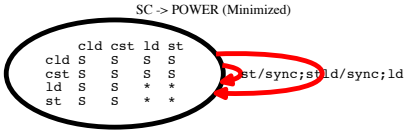


Figure B.10.d: FSM

B.11 SC Upstream, ARM Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	✓S	✓S	✓S	✓S
AC ST	✓S	✓S	✓S	✓S
PO LD	✓S	✓S	✓S	✓S
PO ST	✓S	✓S	✓S	✓S

Figure B.11.a: PPO of (Upstream SC - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS** SS**	ld	dmb; ld	0
0	SSSS SSSS SS** SS**	st	dmb; st	0

Figure B.11.b: FSM Transition Table

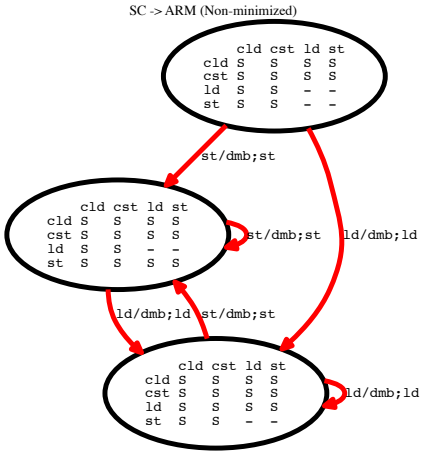


Figure B.11.c: FSM (Pre-minimization)

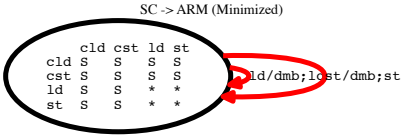


Figure B.11.d: FSM

B.12 TSO Upstream, PLO Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	$\checkmark_S$	—	—	—
PO ST	—	—	—	—

Figure B.12.a: PPO of (Upstream TSO - Downstream PLO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	1
0	SSSS SSSS SS-- SS--	mfence	mfence	0
0	SSSS SSSS SS-- SS--	st	st	0
1	SSSS SSSS SSS- SS--	ld	mlfence; ld	1
1	SSSS SSSS SSS- SS--	mfence	mfence	0
1	SSSS SSSS SSS- SS--	st	st	1

Figure B.12.b: FSM Transition Table

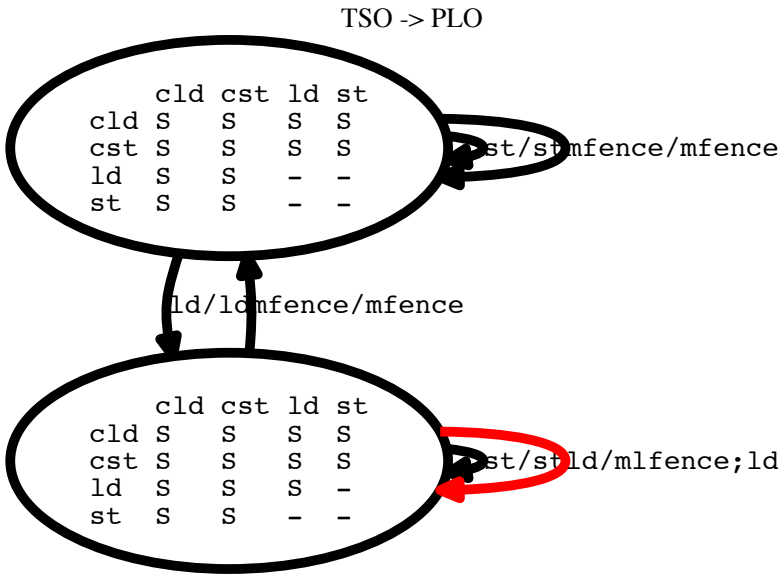


Figure B.12.c: FSM

B.13 TSO Upstream, PSO Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	—	—	$\checkmark_M$	—

Figure B.13.a: PPO of (Upstream TSO - Downstream PSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS-M	ld	ld	0
0	SSSS SSSS SS-- SS-M	mfence	mfence	1
0	SSSS SSSS SS-- SS-M	st	msfence; st	0
1	SSSS SSSS SS-- SS--	ld	ld	1
1	SSSS SSSS SS-- SS--	mfence	mfence	1
1	SSSS SSSS SS-- SS--	st	st	0

Figure B.13.b: FSM Transition Table

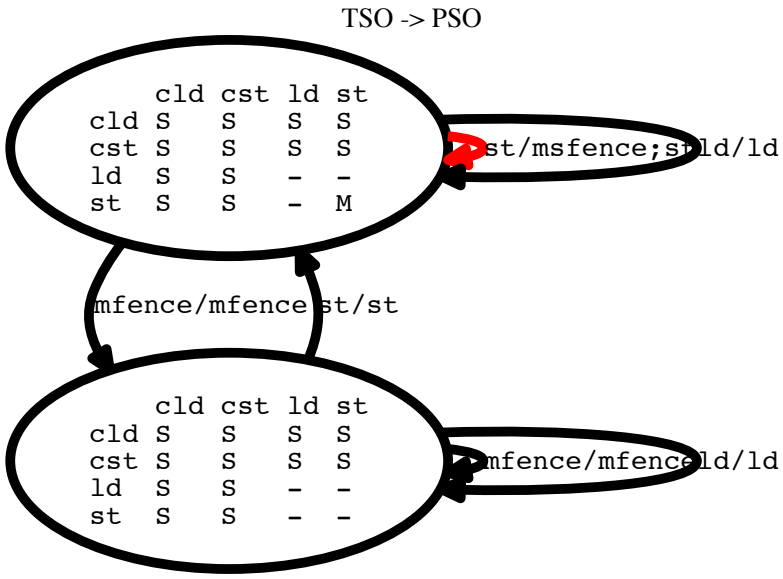


Figure B.13.c: FSM

B.14 TSO Upstream, LSO Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	$\checkmark_S$	—	—	—
PO ST	—	—	$\checkmark_M$	—

Figure B.14.a: PPO of (Upstream TSO - Downstream LSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	3
0	SSSS SSSS SS-- SS--	mfence	mfence	0
0	SSSS SSSS SS-- SS--	st	st	1
1	SSSS SSSS SS-- SS-M	ld	ld	2
1	SSSS SSSS SS-- SS-M	mfence	mfence	0
1	SSSS SSSS SS-- SS-M	st	msfence; st	1
2	SSSS SSSS SSS- SS-M	ld	mlfence; ld	2
2	SSSS SSSS SSS- SS-M	mfence	mfence	0
2	SSSS SSSS SSS- SS-M	st	msfence; st	2
3	SSSS SSSS SSS- SS--	ld	mlfence; ld	3
3	SSSS SSSS SSS- SS--	mfence	mfence	0
3	SSSS SSSS SSS- SS--	st	st	2

Figure B.14.b: FSM Transition Table

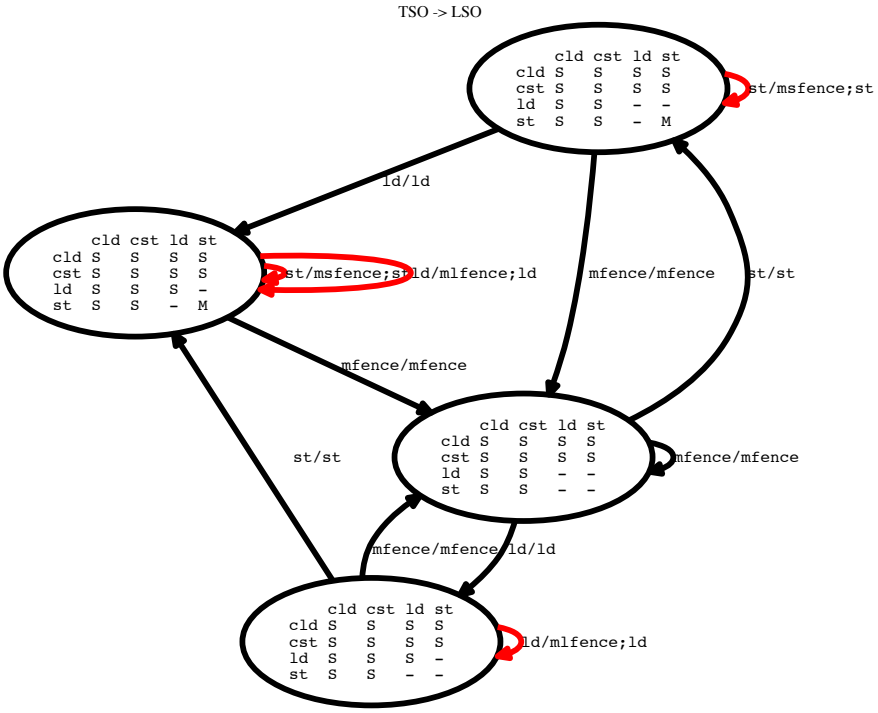


Figure B.14.c: FSM

B.15 TSO Upstream, RMO Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	$\checkmark_S$	—	$\checkmark_S$	—
PO ST	—	—	$\checkmark_M$	—

Figure B.15.a: PPO of (Upstream TSO - Downstream RMO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	2
0	SSSS SSSS SS-- SS--	mfence	mfence	0
0	SSSS SSSS SS-- SS--	st	st	1
1	SSSS SSSS SS-- SS-M	ld	ld	2
1	SSSS SSSS SS-- SS-M	mfence	mfence	0
1	SSSS SSSS SS-- SS-M	st	msfence; st	1
2	SSSS SSSS SSS* SS-*	ld	mlfence; ld	2
2	SSSS SSSS SSS* SS-*	mfence	mfence	0
2	SSSS SSSS SSS* SS-*	st	msfence; st	2

Figure B.15.b: FSM Transition Table

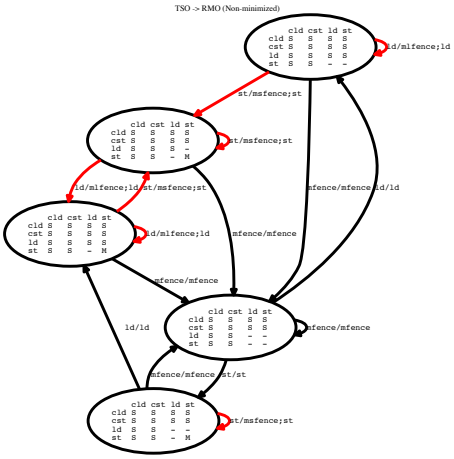


Figure B.15.c: FSM (Pre-minimization)

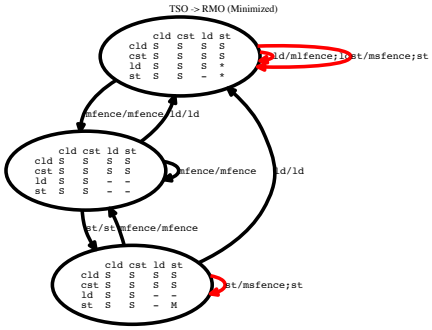


Figure B.15.d: FSM

B.16 TSO Upstream, RMO16 Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	$\checkmark_S$	—	$\checkmark_S$	—
PO ST	—	—	$\checkmark_M$	—

Figure B.16.a: PPO of (Upstream TSO - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SSSS SS--	ld	fence_LL; ld	0
0	SSSS SSSS SSSS SS--	mfence	fence_LL_LS_SL_SS	2
0	SSSS SSSS SSSS SS--	st	fence_LS; st	4
1	SSSS SSSS SSSS SS-M	ld	fence_LL; ld	1
1	SSSS SSSS SSSS SS-M	mfence	fence_LL_LS_SL_SS	2
1	SSSS SSSS SSSS SS-M	st	fence_LS_SS; st	4
2	SSSS SSSS SS-- SS--	ld	ld	0
2	SSSS SSSS SS-- SS--	mfence	fence_LL_LS_SL_SS	2
2	SSSS SSSS SS-- SS--	st	st	3
3	SSSS SSSS SS-- SS-M	ld	ld	1
3	SSSS SSSS SS-- SS-M	mfence	fence_LL_LS_SL_SS	2
3	SSSS SSSS SS-- SS-M	st	fence_SS; st	3
4	SSSS SSSS SSS- SS-M	ld	fence_LL; ld	1
4	SSSS SSSS SSS- SS-M	mfence	fence_LL_LS_SL_SS	2
4	SSSS SSSS SSS- SS-M	st	fence_SS; st	4

Figure B.16.b: FSM Transition Table

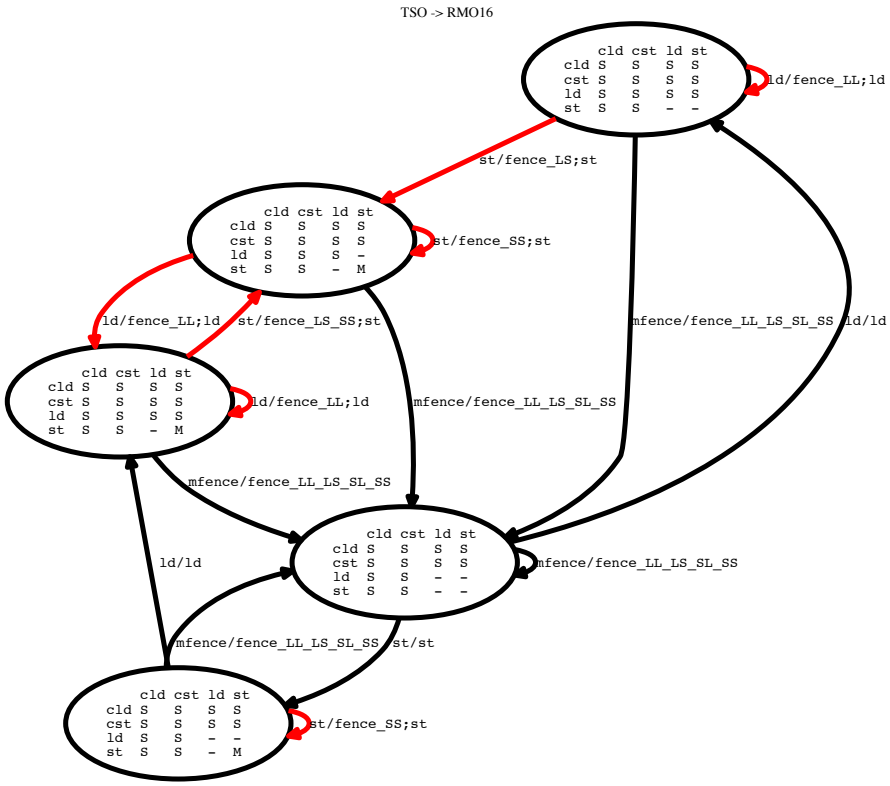


Figure B.16.c: FSM

B.17 TSO Upstream, POWERA Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	$\checkmark_S$	—	$\checkmark_S$	—
PO ST	—	—	$\checkmark_M$	—

Figure B.17.a: PPO of (Upstream TSO - Downstream POWERA)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	3
0	SSSS SSSS SS-- SS--	mfence	sync	0
0	SSSS SSSS SS-- SS--	st	st	1
1	SSSS SSSS SS-- SS-M	ld	ld	2
1	SSSS SSSS SS-- SS-M	mfence	sync	0
1	SSSS SSSS SS-- SS-M	st	sync; st	1
2	SSSS SSSS SSSS SS-M	ld	lwsync; ld	2
2	SSSS SSSS SSSS SS-M	mfence	sync	0
2	SSSS SSSS SSSS SS-M	st	sync; st	1
3	SSSS SSSS SSSS SS--	ld	lwsync; ld	3
3	SSSS SSSS SSSS SS--	mfence	sync	0
3	SSSS SSSS SSSS SS--	st	lwsync; st	1

Figure B.17.b: FSM Transition Table

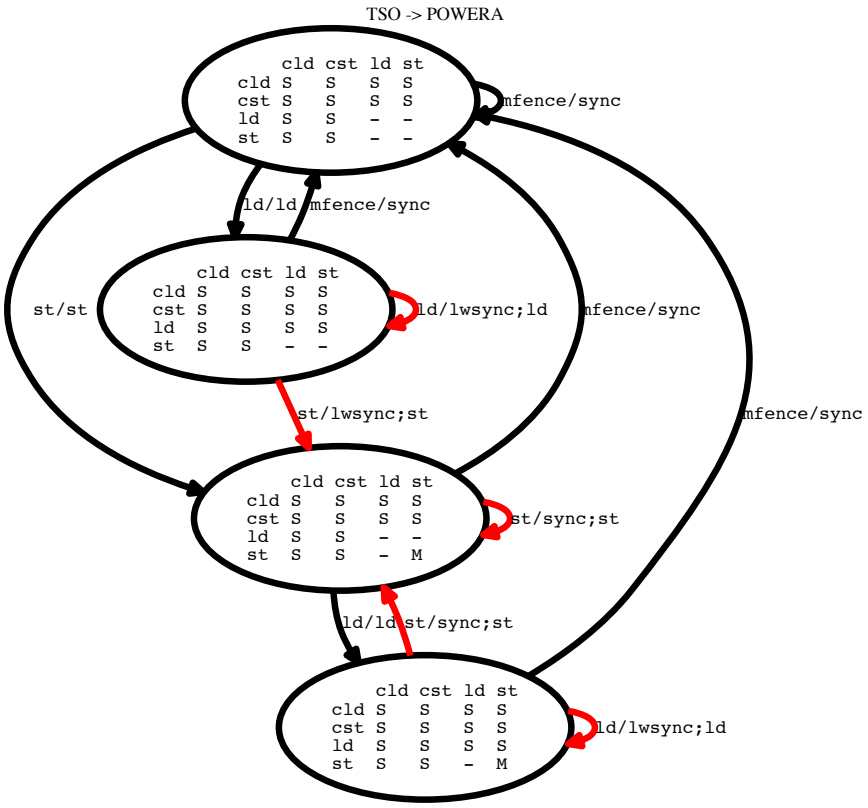


Figure B.17.c: FSM

B.18 TSO Upstream, POWER Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	—	✓$_S$	✓$_M$	✓$_S$

Figure B.18.a: PPO of (Upstream TSO - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS** SS-*	ld	sync; ld	0
0	SSSS SSSS SS** SS-*	mfence	sync	0
0	SSSS SSSS SS** SS-*	st	sync; st	0

Figure B.18.b: FSM Transition Table

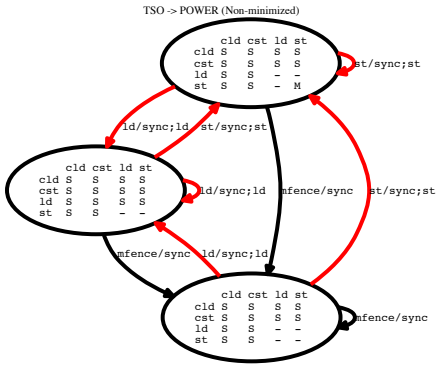


Figure B.18.c: FSM (Pre-minimization)

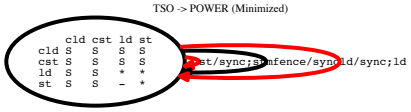


Figure B.18.d: FSM

B.19 TSO Upstream, ARM Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_M$	$\checkmark_S$

Figure B.19.a: PPO of (Upstream TSO - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS** SS-*	ld	dmb; ld	0
0	SSSS SSSS SS** SS-*	mfence	dmb	0
0	SSSS SSSS SS** SS-*	st	dmb; st	0

Figure B.19.b: FSM Transition Table

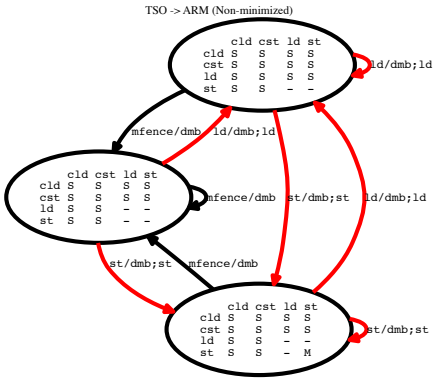


Figure B.19.c: FSM (Pre-minimization)

Figure B.19.d: FSM

B.20 PLO Upstream, PSO Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	—	—	$\checkmark_M$	—

Figure B.20.a: PPO of (Upstream PLO - Downstream PSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS-M	ld	ld	0
0	SSSS SSSS SS-- SS-M	mfence	mfence	1
0	SSSS SSSS SS-- SS-M	st	msfence; st	0
1	SSSS SSSS SS-- SS--	ld	ld	1
1	SSSS SSSS SS-- SS--	mfence	mfence	1
1	SSSS SSSS SS-- SS--	st	st	0

Figure B.20.b: FSM Transition Table

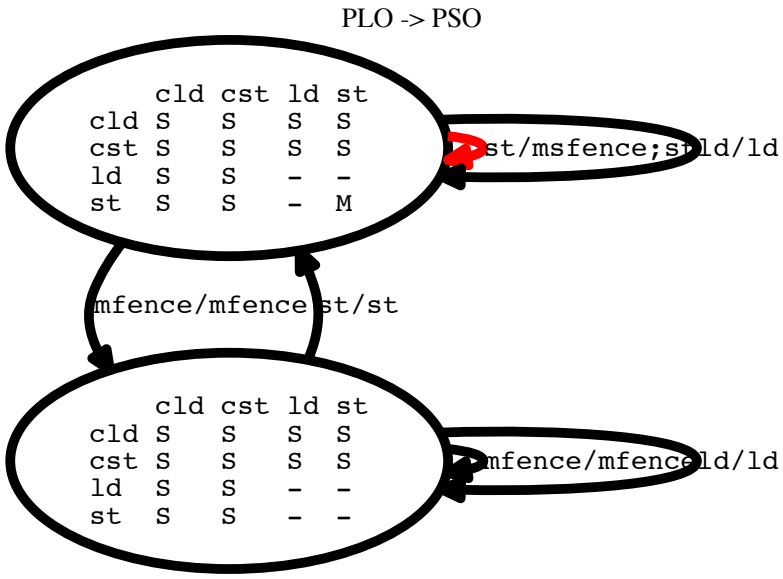


Figure B.20.c: FSM

B.21 PLO Upstream, LSO Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	—	—	$\checkmark_M$	—

Figure B.21.a: PPO of (Upstream PLO - Downstream LSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS-M	ld	ld	0
0	SSSS SSSS SS-- SS-M	mfence	mfence	1
0	SSSS SSSS SS-- SS-M	st	msfence; st	0
1	SSSS SSSS SS-- SS--	ld	ld	1
1	SSSS SSSS SS-- SS--	mfence	mfence	1
1	SSSS SSSS SS-- SS--	st	st	0

Figure B.21.b: FSM Transition Table

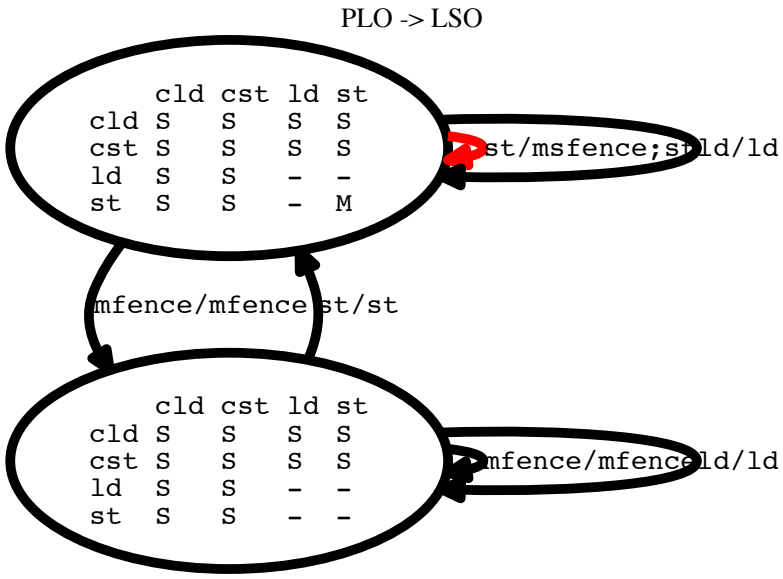


Figure B.21.c: FSM

B.22 PLO Upstream, RMO Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	$\checkmark_S$	—
PO ST	—	—	$\checkmark_M$	—

Figure B.22.a: PPO of (Upstream PLO - Downstream RMO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	1
0	SSSS SSSS SS-- SS--	mfence	mfence	0
0	SSSS SSSS SS-- SS--	st	st	1
1	SSSS SSSS SS-* SS-*	ld	ld	1
1	SSSS SSSS SS-* SS-*	mfence	mfence	0
1	SSSS SSSS SS-* SS-*	st	msfence; st	1

Figure B.22.b: FSM Transition Table

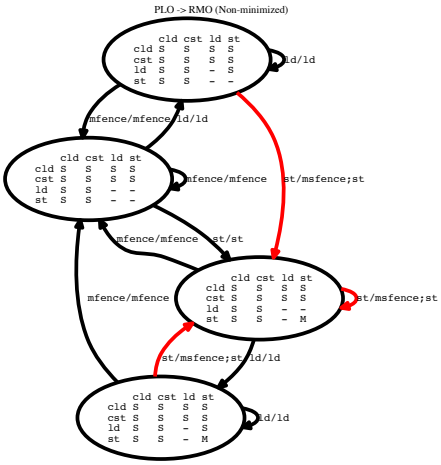


Figure B.22.c: FSM (Pre-minimization)

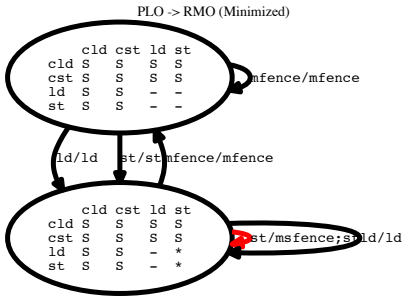


Figure B.22.d: FSM

B.23 PLO Upstream, RMO16 Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	$\checkmark_S$	—
PO ST	—	—	$\checkmark_M$	—

Figure B.23.a: PPO of (Upstream PLO - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS-M	ld	ld	3
0	SSSS SSSS SS-- SS-M	mfence	fence_LL_LS_SL_SS	1
0	SSSS SSSS SS-- SS-M	st	fence_SS; st	0
1	SSSS SSSS SS-- SS--	ld	ld	2
1	SSSS SSSS SS-- SS--	mfence	fence_LL_LS_SL_SS	1
1	SSSS SSSS SS-- SS--	st	st	0
2	SSSS SSSS SS-S SS--	ld	ld	2
2	SSSS SSSS SS-S SS--	mfence	fence_LL_LS_SL_SS	1
2	SSSS SSSS SS-S SS--	st	fence_LS; st	0
3	SSSS SSSS SS-S SS-M	ld	ld	3
3	SSSS SSSS SS-S SS-M	mfence	fence_LL_LS_SL_SS	1
3	SSSS SSSS SS-S SS-M	st	fence_LS_SS; st	0

Figure B.23.b: FSM Transition Table

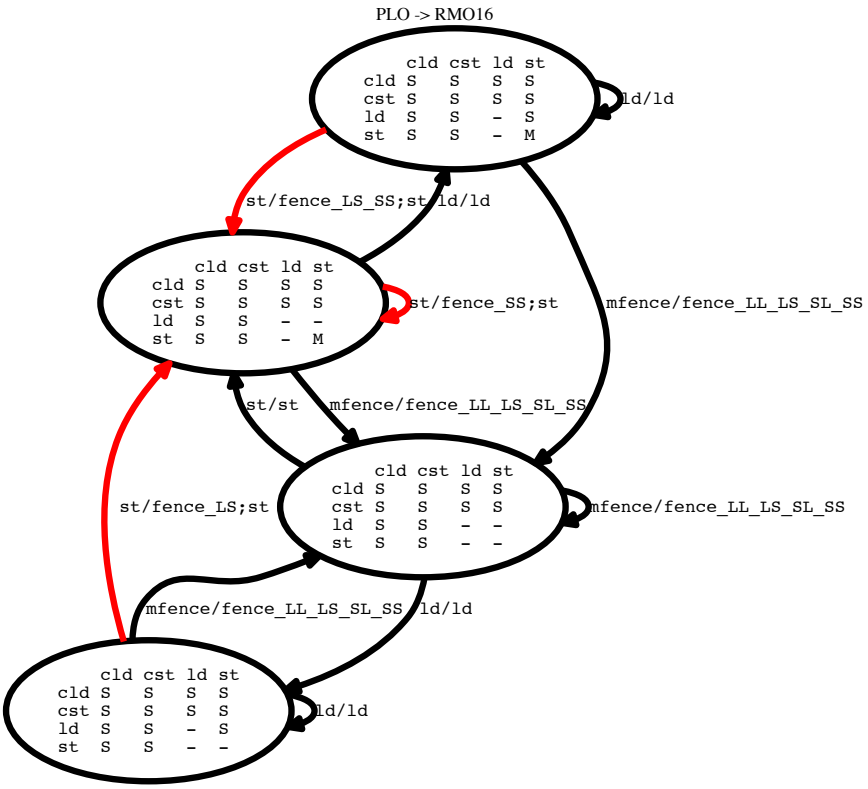


Figure B.23.c: FSM

B.24 PLO Upstream, POWERA Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	$\checkmark_S$	—
PO ST	—	—	$\checkmark_M$	—

Figure B.24.a: PPO of (Upstream PLO - Downstream POWERA)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	2
0	SSSS SSSS SS-- SS--	mfence	sync	0
0	SSSS SSSS SS-- SS--	st	st	1
1	SSSS SSSS SS-* SS-M	ld	ld	1
1	SSSS SSSS SS-* SS-M	mfence	sync	0
1	SSSS SSSS SS-* SS-M	st	sync; st	1
2	SSSS SSSS SS-S SS--	ld	ld	2
2	SSSS SSSS SS-S SS--	mfence	sync	0
2	SSSS SSSS SS-S SS--	st	lwsync; st	1

Figure B.24.b: FSM Transition Table

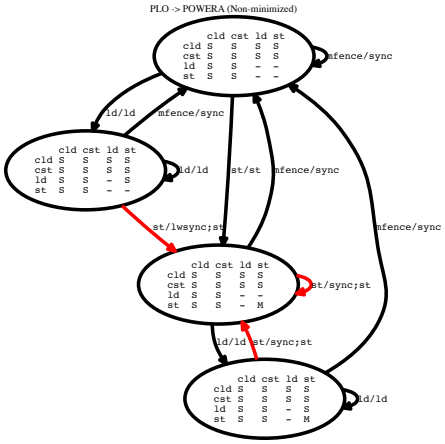


Figure B.24.c: FSM (Pre-minimization)

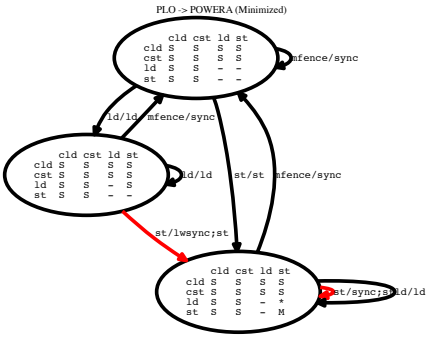


Figure B.24.d: FSM

B.25 PLO Upstream, POWER Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_M$	$\checkmark_S$

Figure B.25.a: PPO of (Upstream PLO - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-* SS-*	ld	sync; ld	0
0	SSSS SSSS SS-* SS-*	mfence	sync	0
0	SSSS SSSS SS-* SS-*	st	sync; st	0

Figure B.25.b: FSM Transition Table

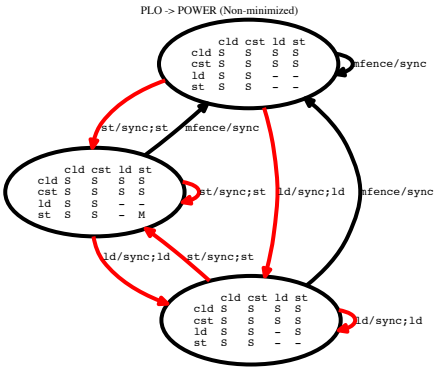


Figure B.25.c: FSM (Pre-minimization)

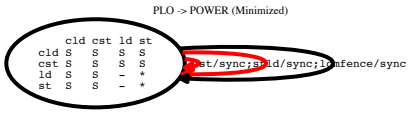


Figure B.25.d: FSM

B.26 PLO Upstream, ARM Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	$\checkmark_M$	$\checkmark_S$

Figure B.26.a: PPO of (Upstream PLO - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-* SS-*	ld	dmb; ld	0
0	SSSS SSSS SS-* SS-*	mfence	dmb	0
0	SSSS SSSS SS-* SS-*	st	dmb; st	0

Figure B.26.b: FSM Transition Table

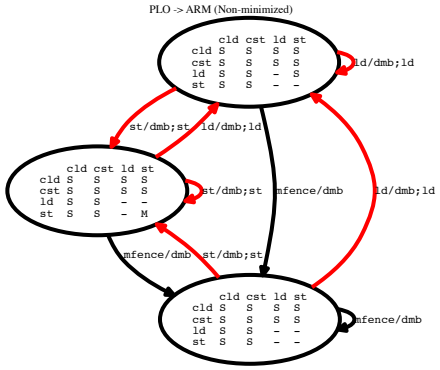


Figure B.26.c: FSM (Pre-minimization)

Figure B.26.d: FSM

B.27 PSO Upstream, PLO Downstream

PPO Diff.	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	$\checkmark_S$	—	—	—
PO ST	—	—	—	—

Figure B.27.a: PPO of (Upstream PSO - Downstream PLO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	1
0	SSSS SSSS SS-- SS--	mfence	mfence	0
0	SSSS SSSS SS-- SS--	st	st	0
1	SSSS SSSS SSS- SS--	ld	mlfence; ld	1
1	SSSS SSSS SSS- SS--	mfence	mfence	0
1	SSSS SSSS SSS- SS--	st	st	1

Figure B.27.b: FSM Transition Table

Figure B.27.c: FSM

B.28 PSO Upstream, LSO Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	$\checkmark_S$	—	—	—
PO ST	—	—	—	—

Figure B.28.a: PPO of (Upstream PSO - Downstream LSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	1
0	SSSS SSSS SS-- SS--	mfence	mfence	0
0	SSSS SSSS SS-- SS--	st	st	0
1	SSSS SSSS SSS- SS--	ld	mlfence; ld	1
1	SSSS SSSS SSS- SS--	mfence	mfence	0
1	SSSS SSSS SSS- SS--	st	st	1

Figure B.28.b: FSM Transition Table

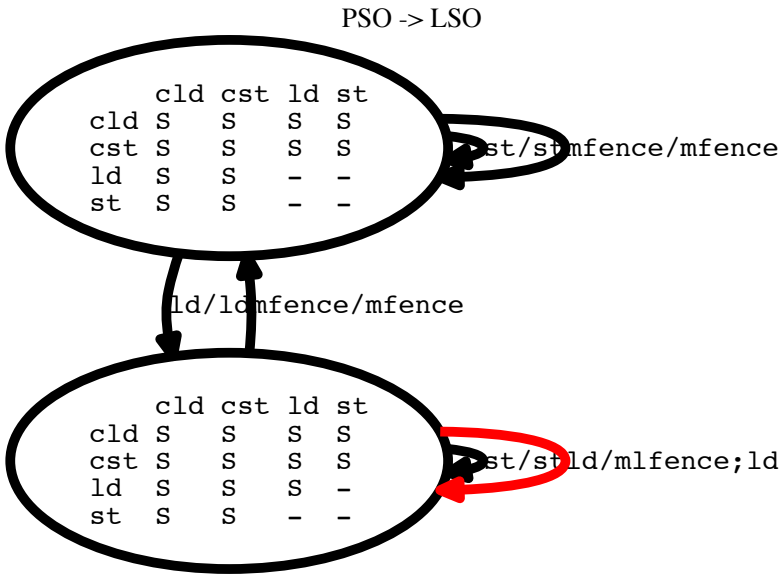


Figure B.28.c: FSM

B.29 PSO Upstream, RMO Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	$\checkmark_S$	—	$\checkmark_S$	—
PO ST	—	—	—	—

Figure B.29.a: PPO of (Upstream PSO - Downstream RMO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SSSS SS--	ld	mlfence; ld	0
0	SSSS SSSS SSSS SS--	mfence	mfence	1
0	SSSS SSSS SSSS SS--	st	msfence; st	2
1	SSSS SSSS SS-- SS--	ld	ld	0
1	SSSS SSSS SS-- SS--	mfence	mfence	1
1	SSSS SSSS SS-- SS--	st	st	1
2	SSSS SSSS SSS- SS--	ld	mlfence; ld	0
2	SSSS SSSS SSS- SS--	mfence	mfence	1
2	SSSS SSSS SSS- SS--	st	st	2

Figure B.29.b: FSM Transition Table

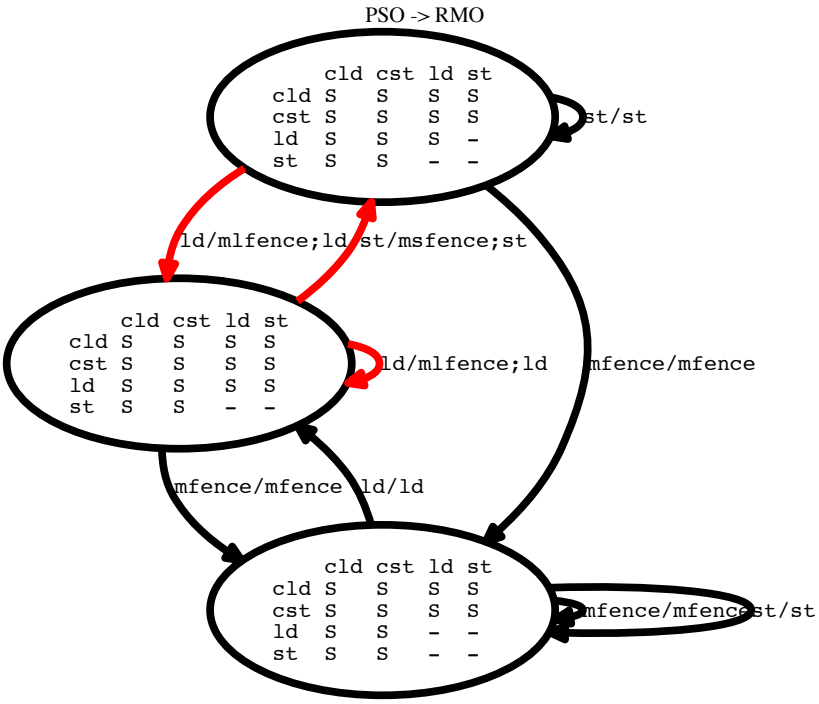


Figure B.29.c: FSM

B.30 PSO Upstream, RMO16 Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	$\checkmark_S$	—	$\checkmark_S$	—
PO ST	—	—	—	—

Figure B.30.a: PPO of (Upstream PSO - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SSSS SS--	ld	fence_LL; ld	0
0	SSSS SSSS SSSS SS--	mfence	fence_LL_LS_SL_SS	1
0	SSSS SSSS SSSS SS--	st	fence_LS; st	2
1	SSSS SSSS SS-- SS--	ld	ld	0
1	SSSS SSSS SS-- SS--	mfence	fence_LL_LS_SL_SS	1
1	SSSS SSSS SS-- SS--	st	st	1
2	SSSS SSSS SSS- SS--	ld	fence_LL; ld	0
2	SSSS SSSS SSS- SS--	mfence	fence_LL_LS_SL_SS	1
2	SSSS SSSS SSS- SS--	st	st	2

Figure B.30.b: FSM Transition Table

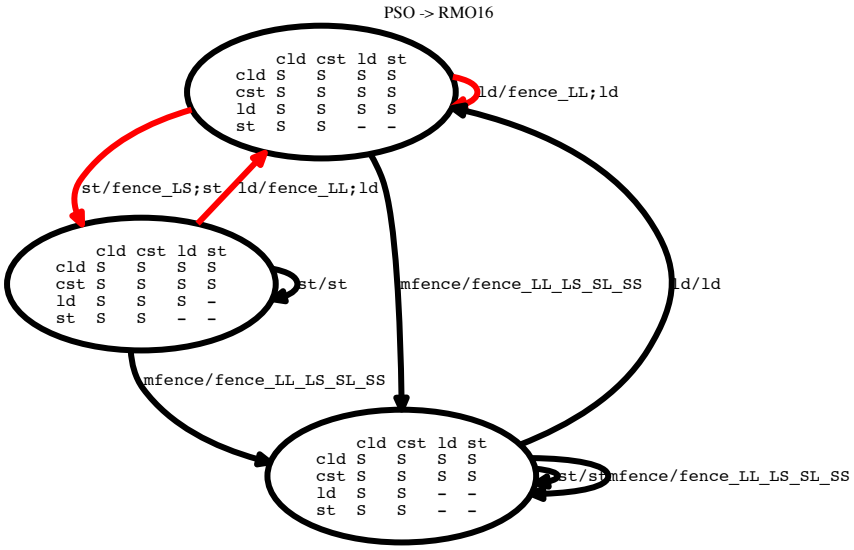


Figure B.30.c: FSM

B.31 PSO Upstream, POWERA Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	$\checkmark_S$	—	$\checkmark_S$	—
PO ST	—	—	—	—

Figure B.31.a: PPO of (Upstream PSO - Downstream POWERA)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SSSS SS--	ld	lwsync; ld	0
0	SSSS SSSS SSSS SS--	mfence	sync	1
0	SSSS SSSS SSSS SS--	st	lwsync; st	1
1	SSSS SSSS SS-- SS--	ld	ld	0
1	SSSS SSSS SS-- SS--	mfence	sync	1
1	SSSS SSSS SS-- SS--	st	st	1

Figure B.31.b: FSM Transition Table

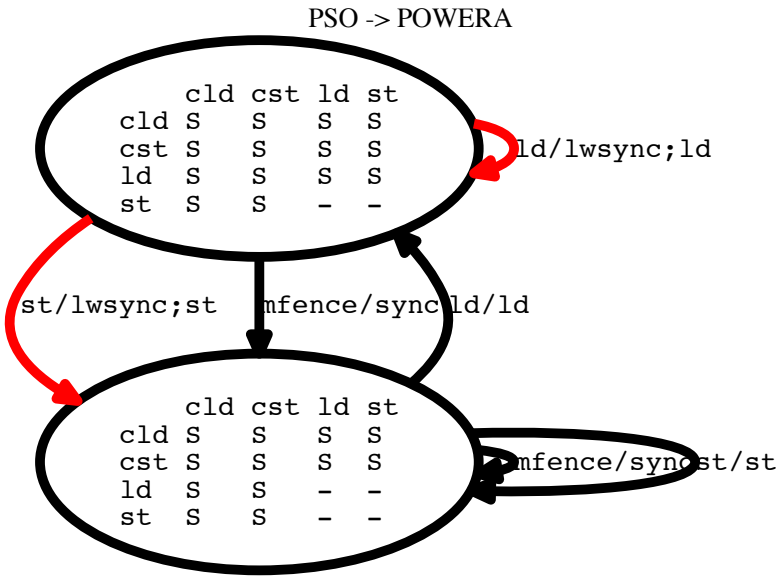


Figure B.31.c: FSM

B.32 PSO Upstream, POWER Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	—	✓$_S$	—	✓$_S$

Figure B.32.a: PPO of (Upstream PSO - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS** SS--	ld	sync; ld	0
0	SSSS SSSS SS** SS--	mfence	sync	0
0	SSSS SSSS SS** SS--	st	sync; st	0

Figure B.32.b: FSM Transition Table

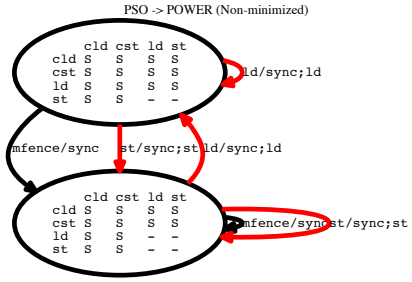


Figure B.32.c: FSM (Pre-minimization)

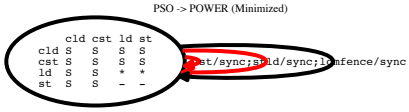


Figure B.32.d: FSM

B.33 PSO Upstream, ARM Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO ST	—	✓$_S$	—	✓$_S$

Figure B.33.a: PPO of (Upstream PSO - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS** SS--	ld	dmb; ld	0
0	SSSS SSSS SS** SS--	mfence	dmb	0
0	SSSS SSSS SS** SS--	st	dmb; st	0

Figure B.33.b: FSM Transition Table

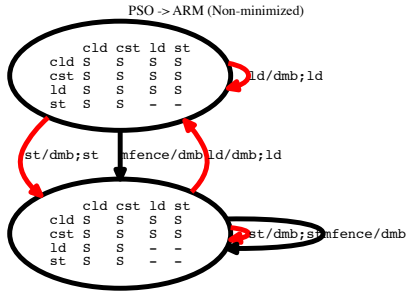


Figure B.33.c: FSM (Pre-minimization)

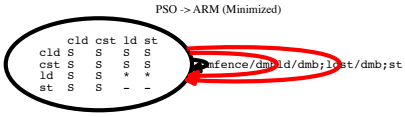


Figure B.33.d: FSM

B.34 LSO Upstream, RMO Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	$\checkmark_S$	—
PO ST	—	—	—	—

Figure B.34.a: PPO of (Upstream LSO - Downstream RMO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	1
0	SSSS SSSS SS-- SS--	mfence	mfence	0
0	SSSS SSSS SS-- SS--	st	st	0
1	SSSS SSSS SS-S SS--	ld	ld	1
1	SSSS SSSS SS-S SS--	mfence	mfence	0
1	SSSS SSSS SS-S SS--	st	msfence; st	0

Figure B.34.b: FSM Transition Table

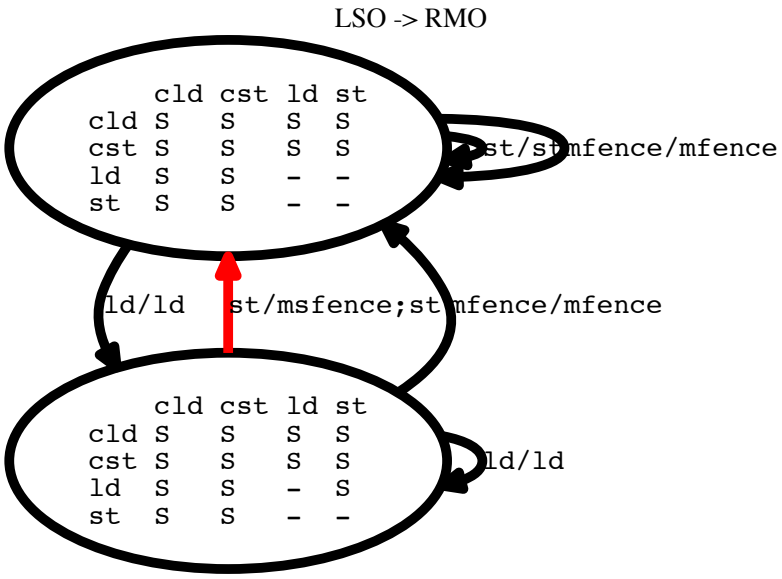


Figure B.34.c: FSM

B.35 LSO Upstream, RMO16 Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	✓$_S$	—
PO ST	—	—	—	—

Figure B.35.a: PPO of (Upstream LSO - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	1
0	SSSS SSSS SS-- SS--	mfence	fence_LL_LS_SL_SS	0
0	SSSS SSSS SS-- SS--	st	st	0
1	SSSS SSSS SS-S SS--	ld	ld	1
1	SSSS SSSS SS-S SS--	mfence	fence_LL_LS_SL_SS	0
1	SSSS SSSS SS-S SS--	st	fence_LS; st	0

Figure B.35.b: FSM Transition Table

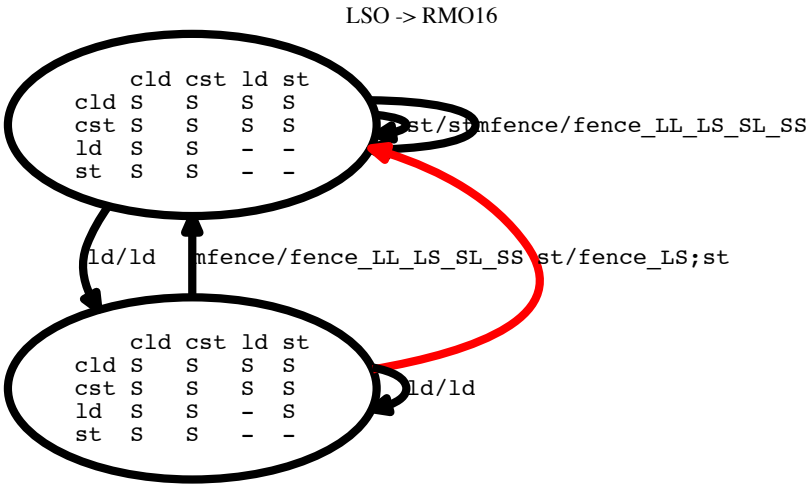


Figure B.35.c: FSM

B.36 LSO Upstream, POWERA Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	✓$_S$	—
PO ST	—	—	—	—

Figure B.36.a: PPO of (Upstream LSO - Downstream POWERA)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	1
0	SSSS SSSS SS-- SS--	mfence	sync	0
0	SSSS SSSS SS-- SS--	st	st	0
1	SSSS SSSS SS-S SS--	ld	ld	1
1	SSSS SSSS SS-S SS--	mfence	sync	0
1	SSSS SSSS SS-S SS--	st	lwsync; st	0

Figure B.36.b: FSM Transition Table

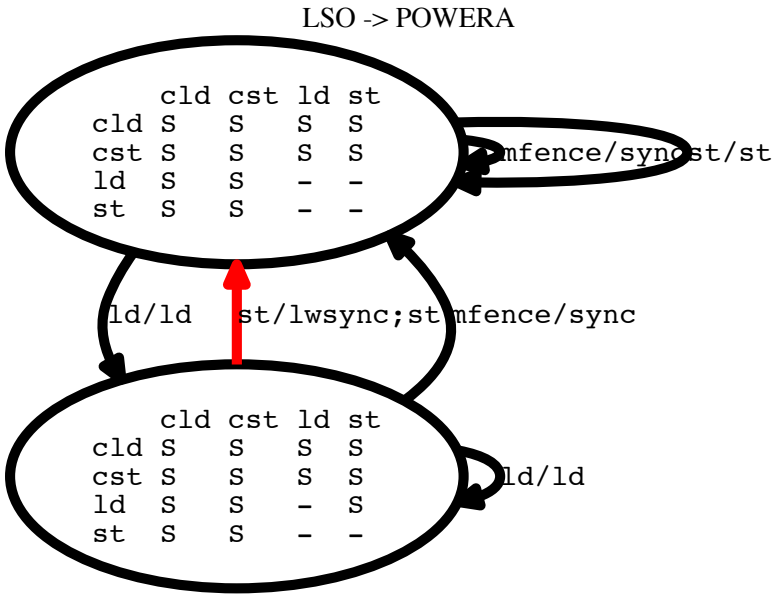


Figure B.36.c: FSM

B.37 LSO Upstream, POWER Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO ST	—	$\checkmark_S$	—	$\checkmark_S$

Figure B.37.a: PPO of (Upstream LSO - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-* SS--	ld	sync; ld	0
0	SSSS SSSS SS-* SS--	mfence	sync	0
0	SSSS SSSS SS-* SS--	st	sync; st	0

Figure B.37.b: FSM Transition Table

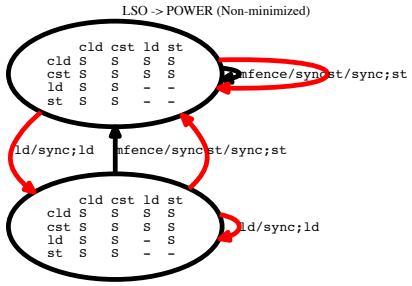


Figure B.37.c: FSM (Pre-minimization)

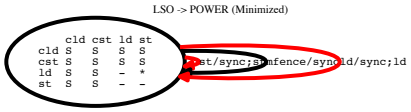


Figure B.37.d: FSM

B.38 LSO Upstream, ARM Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	✓S	✓S	✓S	✓S
AC ST	✓S	✓S	✓S	✓S
PO LD	—	✓S	✓S	✓S
PO ST	—	✓S	—	✓S

Figure B.38.a: PPO of (Upstream LSO - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-* SS--	ld	dmb; ld	0
0	SSSS SSSS SS-* SS--	mfence	dmb	0
0	SSSS SSSS SS-* SS--	st	dmb; st	0

Figure B.38.b: FSM Transition Table

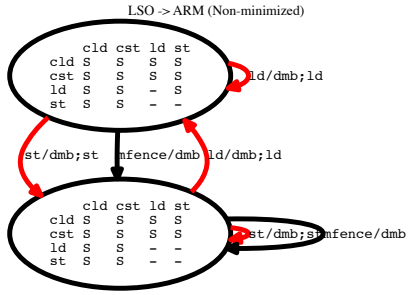


Figure B.38.c: FSM (Pre-minimization)

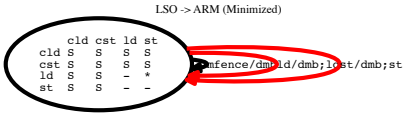


Figure B.38.d: FSM

B.39 RMO Upstream, RMO16 Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	—	—	—	—

Figure B.39.a: PPO of (Upstream RMO - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	0
0	SSSS SSSS SS-- SS--	mfence	fence_LL_LS_SL_SS	0
0	SSSS SSSS SS-- SS--	st	st	0

Figure B.39.b: FSM Transition Table

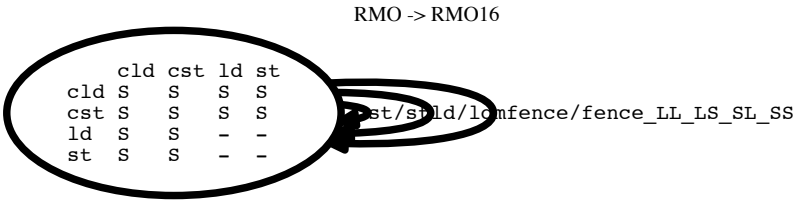


Figure B.39.c: FSM

B.40 RMO Upstream, POWERA Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	—	—	—	—

Figure B.40.a: PPO of (Upstream RMO - Downstream POWERA)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	0
0	SSSS SSSS SS-- SS--	mfence	sync	0
0	SSSS SSSS SS-- SS--	st	st	0

Figure B.40.b: FSM Transition Table

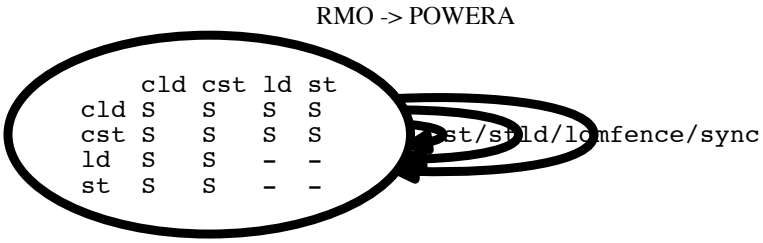


Figure B.40.c: FSM

B.41 RMO Upstream, POWER Downstream

PPO Diff.				
	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	—	$\checkmark_S$
PO ST	—	$\checkmark_S$	—	$\checkmark_S$

Figure B.41.a: PPO of (Upstream RMO - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	sync; ld	0
0	SSSS SSSS SS-- SS--	mfence	sync	0
0	SSSS SSSS SS-- SS--	st	sync; st	0

Figure B.41.b: FSM Transition Table

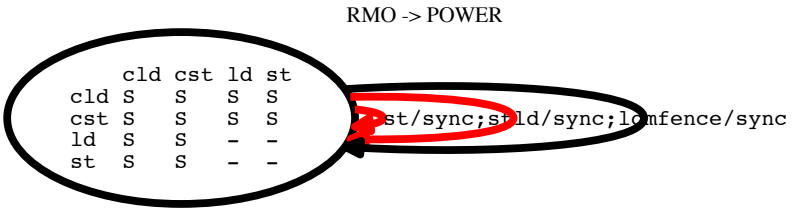


Figure B.41.c: FSM

B.42 RMO Upstream, ARM Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	—	$\checkmark_S$
PO ST	—	$\checkmark_S$	—	$\checkmark_S$

Figure B.42.a: PPO of (Upstream RMO - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	dmb; ld	0
0	SSSS SSSS SS-- SS--	mfence	dmb	0
0	SSSS SSSS SS-- SS--	st	dmb; st	0

Figure B.42.b: FSM Transition Table

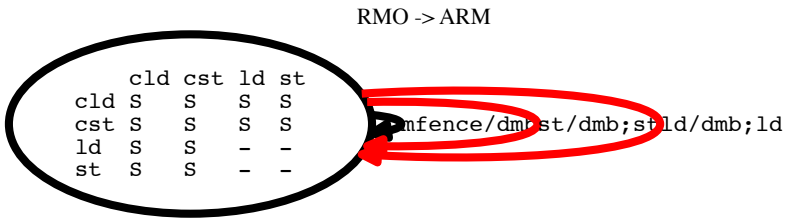


Figure B.42.c: FSM

B.43 POWERA Upstream, RMO Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	—	—	—	—

Figure B.43.a: PPO of (Upstream POWERA - Downstream RMO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	0
0	SSSS SSSS SS-- SS--	lwsync	mfence	0
0	SSSS SSSS SS-- SS--	sync	mfence	0
0	SSSS SSSS SS-- SS--	st	st	0

Figure B.43.b: FSM Transition Table

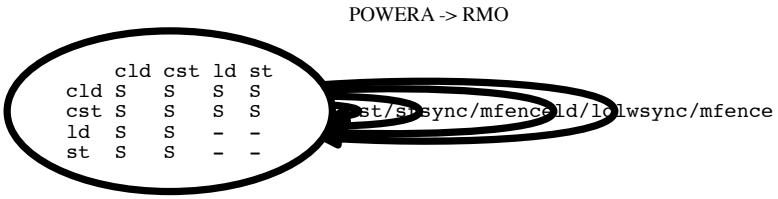


Figure B.43.c: FSM

B.44 POWERA Upstream, RMO16 Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	—	—	—	—

Figure B.44.a: PPO of (Upstream POWERA - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	0
0	SSSS SSSS SS-- SS--	lwsync	fence_LL_LS_SS	0
0	SSSS SSSS SS-- SS--	sync	fence_LL_LS_SL_SS	0
0	SSSS SSSS SS-- SS--	st	st	0

Figure B.44.b: FSM Transition Table

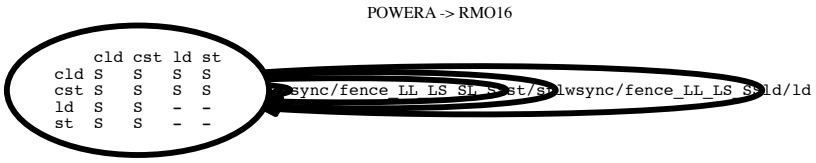


Figure B.44.c: FSM

B.45 POWERA Upstream, POWER Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	✓$_S$	✓$_S$	✓$_S$	✓$_S$
AC ST	✓$_S$	✓$_S$	✓$_S$	✓$_S$
PO LD	—	✓$_S$	—	✓$_S$
PO ST	—	✓$_S$	—	✓$_S$

Figure B.45.a: PPO of (Upstream POWERA - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	sync; ld	0
0	SSSS SSSS SS-- SS--	lwsync	sync	0
0	SSSS SSSS SS-- SS--	sync	sync	0
0	SSSS SSSS SS-- SS--	st	sync; st	0

Figure B.45.b: FSM Transition Table

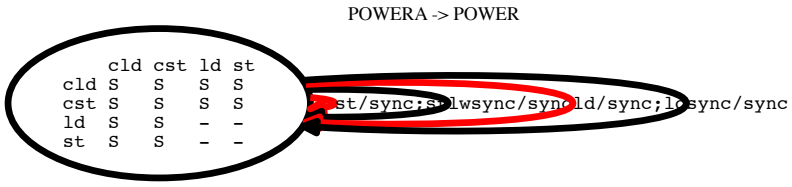


Figure B.45.c: FSM

B.46 POWERA Upstream, ARM Downstream

PPO Diff.

	PO LD	BC LD	PO ST	BC ST
AC LD	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
AC ST	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$	$\checkmark_S$
PO LD	—	$\checkmark_S$	—	$\checkmark_S$
PO ST	—	$\checkmark_S$	—	$\checkmark_S$

Figure B.46.a: PPO of (Upstream POWERA - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	dmb; ld	0
0	SSSS SSSS SS-- SS--	lwsync	dmb	0
0	SSSS SSSS SS-- SS--	sync	dmb	0
0	SSSS SSSS SS-- SS--	st	dmb; st	0

Figure B.46.b: FSM Transition Table

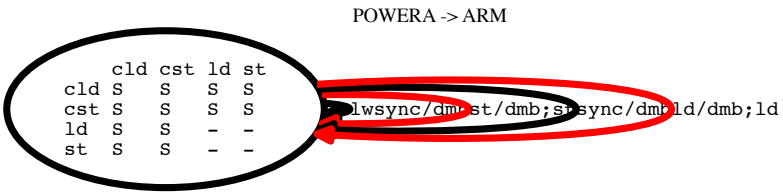


Figure B.46.c: FSM

B.47 POWER Upstream, ARM Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	—	—	—	—

Figure B.47.a: PPO of (Upstream POWER - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	---- ---- ---- ----	ld	ld	0
0	---- ---- ---- ----	lwsync	dmb	0
0	---- ---- ---- ----	sync	dmb	0
0	---- ---- ---- ----	st	st	0

Figure B.47.b: FSM Transition Table

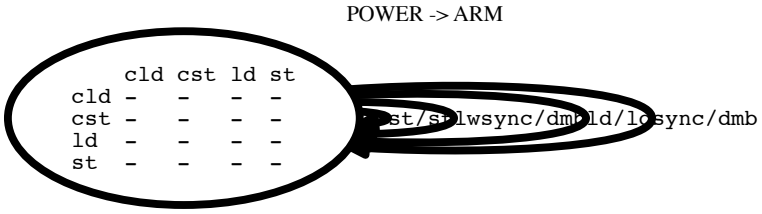


Figure B.47.c: FSM

B.48 ARM Upstream, POWER Downstream

	PPO Diff.			
	PO LD	BC LD	PO ST	BC ST
AC LD	—	—	—	—
AC ST	—	—	—	—
PO LD	—	—	—	—
PO ST	—	—	—	—

Figure B.48.a: PPO of (Upstream ARM - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	---- ---- ---- ----	dmb	sync	0
0	---- ---- ---- ----	ld	ld	0
0	---- ---- ---- ----	st	st	0

Figure B.48.b: FSM Transition Table

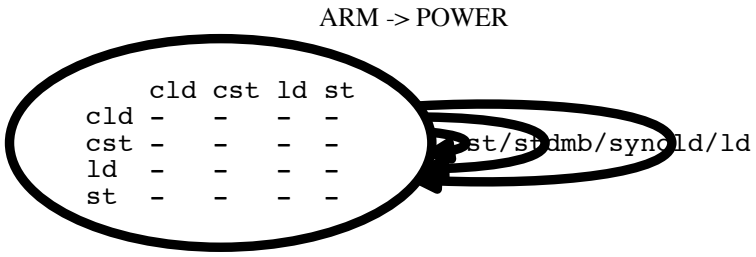


Figure B.48.c: FSM

Bibliography

[AAS03] Allon Adir, Hagit Attiya, and Gil Shurek. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 14(5):502–515, 2003.

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. *University of California at Berkeley Technical Report No. UCB/EECS-2006-183*, December 2006.

[ABD+15] Jade Alglave, Mark Batty, Alastair Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: weak behaviours and programming assumptions. *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[ADC11] Thomas J. Ashby, Pedro Diaz, and Marcelo Cintra. Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters. *IEEE Transactions on Computers*, 60(4):472–483, 2011.

[Adv93] Sarita V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, December 1993. Technical Report 1198.

[AFI+09] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of Power and ARM machine code. *Declarative Aspects of Multicore Programming (DAMP) Workshop, in conjunction with POPL*, 2009.

[AG96] Sarita Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. *17th International Symposium on Computer Architecture (ISCA)*, 1990.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. *25th International Conference on Computer Aided Verification (CAV)*, 2013.

[Alg12] Jade Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design (FMSD)*, 41(2):178–210, 2012.

[AM06] Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. *33rd International Symposium on Computer Architecture (ISCA)*, 2006.

[AMD11] AMD. Revision guide for AMD family 10h processors. `http://developer.amd.com/wordpress/media/2012/10/41322.pdf`, August 2011.

[AMD12] AMD. Southern Islands series instruction set architecture, rev. 1.1. `http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf`, December 2012.

[AMD13] AMD. AMD64 architecture programmer's manual, rev. 3.24. `http://developer.amd.com/resources/documentation-articles/developer-guides-manuals`, Oct. 2013.

[AMP96] Rajeev Alur, Ken McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. In *11th Symposium on Logic in Computer Science (LICS)*, 1996.

[AMS+12] Jade Alglave, Luc Maranget, Susmit Sarkar, et al. diy, a tool suite for testing shared memory models. `http://diy.inria.fr`, 2012.

[AMSS10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. *22nd International Conference on Computer Aided Verification (CAV)*, 2010.

[AMSS11] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 41–44. Springer, 2011.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):7:1–7:74, July 2014.

[ARM09] ARM. Barrier litmus tests and cookbook, 2009.

[ARM11] ARM. Cortex-A9 MPCore™ programmer advice notice: Read-after-read hazards, ARM reference 761319, 2011.

[ARM13] ARM. ARM architecture reference manual, 2013.

[AŠ07] David Aspinall and Jaroslav Ševčík. Java memory model examples: Good, bad and ugly. *1st International Workshop on Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, 2007.

[BA08] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. *29th Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[BAM07] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. *28th Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[Bat04] Mark J. Batty. *The C11 and C++11 Concurrency Model*. PhD thesis, University of Cambridge, November 2004.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, May 2011.

[BC11] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[BC13] Thomas Braibant and Adam Chlipala. Formal verification of hardware synthesis. *25th International Conference on Computer Aided Verification (CAV)*, 2013.

[BCC+10] Moshe Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. Analyzing parallel programs with Pin. *IEEE Computer*, 43(3):34–41, 2010.

[BCH03] Jesse D. Bingham, Anne E. Condon, and Alan J. Hu. Toward a decidable notion of sequential consistency. In *15th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2003.

[BDE+03] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. *36th International Symposium on Microarchitecture (MICRO)*, 2003.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[BJK+06] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together–formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4-5):411–430, 2006.

[BLM11] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level TLBs for chip multiprocessors. *17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[BMO+12] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. *39th Symposium on Principles of Programming Languages (POPL)*, 2012.

[BMW09] Colin Blundell, Milo Martin, and Thomas Wenisch. InvisiFence: Performance-transparent memory ordering in conventional multiprocessors. *36th International Symposium on Computer Architecture (ISCA)*, 2009.

[BOS+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. *38th Symposium on Principles of Programming Languages (POPL)*, 2011.

[Bro02] Broadcom. Migrating CPU specific code from the PowerPC to the Broadcom SB-1 processor. *White Paper SB-1-WP100-R*, 2002.

[BSDA14] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified compilation for shared-memory C. *Programming Languages and Systems*, pages 107–127, 2014.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, chapter 26, pages 825–885. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009.

[Cen15] Centre for Computing History. The Manchester Baby, the world's first stored program computer, ran its first program. `http://www.computinghistory.org.uk/det/6013/The-Manchester-Baby-the-world-s-first-stored-program-computer-ran-its-` 2015.

[CGH+93] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the Futurebus+ cache coherence protocol. In *International Conference on Computer Hardware Description Languages and their Applications (CHDL)*, pages 15–30, 1993.

[CGJ04] Solange Coupet-Grimal and Line Jakubiec. Certifying circuits in type theory. *Formal Aspects of Computing*, 16(4):352–373, 2004.

[CH03] Anne E. Condon and Alan J. Hu. Automatable verification of sequential consistency. *Theory of Computing Systems*, 36(5):431–460, 2003.

[CKS+11] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[CL02] Harold W. Cain and Mikko H. Lipasti. Verifying sequential consistency using vector clocks. *14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2002.

[CLH+09] Yunji Chen, Yi Lv, Weiwu Hu, Tianshi Chen, Haihua Shen, Pengyu Wang, and Hong Pan. Fast complete memory consistency verification. In *15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[CLN03] Harold W. Cain, Mikko H. Lipasti, and Ravi Nair. Constraint graph analysis of multithreaded programs. *12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003.

[CLS03] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. The complexity of verifying memory coherence. In *15th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 254–255. ACM, 2003.

[CMP04] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 382–398. Springer, 2004.

[CMP08] Kaiyu Chen, Sharad Malik, and Priyadarsan Patra. Runtime validation of memory ordering using constraint graph checking. In *14th International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[Col92] William W. Collier. *Reasoning about Parallel Architectures*. Prentice Hall, 1992.

[Com15] Computer History Museum. Timeline of computer history. `http://www.computerhistory.org/timeline`, 2015.

[CRDI07] Thomas Chen, Ram Raghavan, Jason N. Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementation–a performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[CTMT07] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk enforcement of sequential consistency. *34th International Symposium on Computer Architecture (ISCA)*, 2007.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. *11th International Conference on Computer Design (ICCD)*, 1992.

[DGB+03] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta Code Morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *1st International Symposium on Code Generation and Optimization (CGO)*, 2003.

[DGY+74] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits (JSSC)*, 9(5):256–268, 1974.

[Dig92] Digital Equipment Corporation. Alpha architecture reference manual. 1992.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DMT13] Yuelu Duan, Abdullah Muzahid, and Josep Torrellas. WeeFence: Toward making fences free in TSO. *40th International Symposium on Computer Architecture (ISCA)*, 2013.

[DSB86] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. *13th International Symposium on Computer Architecture (ISCA)*, 1986.

[DVT12] Matthew DeVuyst, Ashish Venkat, and Dean Tullsen. Execution migration in a heterogeneous-ISA chip multiprocessor. *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[Elv] Marco Elver. gem5 source repository, changeset 10149. `http://repo.gem5.org/gem5/rev/45a67d84fd4a`.

[EM] J. Presper Eckert and John W. Mauchly. Electronic numerical integrator and computer. US Patent 3,120,606, filed June 26, 1947, issued February 4 1964; invalidated 19 October 1973.

[EN14] Marco Elver and Vijay Nagarajan. TSO-CC: consistency directed cache coherence for TSO. *20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[GEAS00] Michael Gschwind, Kemal Ebcioğlu, Erik Altman, and Sumedh Sathaye. Binary translation and architecture convergence issues for IBM System/390. *ICS*, 2000.

[GGH91] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. *29th International Conference on Parallel Processing (ICPP)*, 1991.

[Gha95] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, December 1995. Technical Report CSL-TR-95-685.

[GK92] Phillip B. Gibbons and Ephraim Korach. The complexity of sequential consistency. In *4th Symposium on Parallel and Distributed Processing (IPDPS)*, 1992.

[GK94] Phillip B. Gibbons and Ephraim Korach. On testing cache-coherent shared memories. In *6th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1994.

[GK97] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *17th International Symposium on Computer Architecture (ISCA)*, 1990.

[Goo15] Google. Android. `http://www.android.com`, 2015.

[Gre11] Peter Greenhalgh. big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. *ARM White Paper*, 2011.

[GSSVD00] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Van Doren. Architecture and design of AlphaServer GS320. *SIGPLAN Notices*, 35(11):13–24, November 2000.

[HJ06] Lisa Higham and LillAnne Jackson. Translating between Itanium and Sparc memory consistency models. *18th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2006.

[HM08] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, (7):33–38, 2008.

[HP11] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[HR07] Thuan Quang Huynh and Abhik Roychoudhury. Memory model sensitive bytecode verification. *Formal Methods in System Design (FMSD)*, 31, 2007.

[HS13] Blake Hechtman and Daniel Sorin. Exploring memory consistency for massively-threaded throughput-oriented processors. *40th International Symposium on Computer Architecture (ISCA)*, 2013.

[HS15] Brett Howse and Ryan Smith. Tick tock on the rocks: Intel delays 10nm, adds 3rd gen 14nm Core product "Kaby Lake". `http://www.anandtech.com/show/9447/intel-10nm-and-kaby-lake`, July 2015.

[HSA13] HSA Foundation. HSA programmer reference manual specification 1.0 provisional. `http://www.hsafoundation.com/standards`, May 2013.

[HVML04] Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, and Juin-Yeu Joseph Lu. TSOtool: A program for verifying memory systems using the memory consistency model. *31st International Symposium on Computer Architecture (ISCA)*, 2004.

[IBM83] IBM. IBM System 370: Principles of operation. May 1983.

[IBM13] IBM. Power ISA version 2.07, 2013.

[Inc04] Bluespec Inc. Bluespec SystemVerilog version 3.8 reference guide, November 2004.

[Int07] Intel. Intel 64 architecture memory ordering white paper. 2007. SKU 318147-001.

[Int10] Intel. Intel Itanium architecture software developer's manual, revision 2.3. 2010.

[Int13a] Intel. Intel 64 and IA-32 architectures software developer's manual. *Order Number 325462-048US*, Sept. 2013.

[Int13b] Intel. Intel Itanium architecture software developer's manual, rev. 2.3. 2013.

[Int15] Intel. Intel Xeon processor E3-1200 v3 product family, specification update. `http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf`, April 2015.

[ISO11a] ISO/IEC. Information technology – Programming languages – C++. 2011. ISO/IEC 14882.

[ISO11b] ISO/IEC. Information technology – Programming languages – C. 2011. ISO/IEC 9899.

[KCA92] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the window of vulnerability in multiphase memory transactions. *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

[Khr] Khronos Group. OpenCL 2.0. `http://www.khronos.org/opencl`.

[KM08] Stefanos Kaxiras and Margaret Martonosi. *Computer architecture techniques for power-efficiency*. Number 4 in Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2008.

[Knu97] Donald Knuth. Notes on the van Emde Boas construction of priority deques: an innovative use of recursion. *Classroom notes, Stanford University*, March 1997.

[KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *21st International Symposium on Computer Architecture (ISCA)*, 1994.

[KSB95] Leonidas Kontothanassis, Michael L. Scott, and Ricardo Bianchini. Lazy release consistency for hardware-coherent multiprocessors. *International Conference on Supercomputing*, 1995.

[KVY10] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. *10th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2010.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. *2nd International Symposium on Code Generation and Optimization (CGO)*, 2004.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computing*, C-28, Sept. 1979.

[Lam02] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[LBM13] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10, April 2013.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *26th Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[LGCP13] Nhat Minh Lê, Adrien Guatto, Albert Cohen, and Antoniu Pop. Correct and efficient bounded FIFO queues. *23rd International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*, 2013.

[LM13] Daniel Lustig and Margaret Martonosi. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. *19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[LP00] Jaejin Lee and David A. Padua. Hiding relaxed memory consistency with compilers. *9th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2000.

[LPCZN13] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. *18th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[LPM14] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. *47th International Symposium on Microarchitecture (MICRO)*, 2014.

[LPM15] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Verifying correct microarchitectural enforcement of memory consistency models. *IEEE Micro (Top Picks of 2014)*, 35(3), 2015.

[LSMB15] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. MEMPATROL: Verifying memory ordering at the hardware-OS interface. *under submission*, 2015.

[LTPM15a] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. ArMOR: Defending against consistency model mismatches in heterogeneous architectures. *42nd International Symposium on Computer Architecture (ISCA)*, 2015.

[LTPM15b] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. ArMOR: Defending against consistency model mismatches in heterogeneous architectures. *Princeton Computer Science Technical Report (number TR-981-15)*, 2015. (extended version of conference paper).

[Lus14] Daniel Lustig. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. `https://github.com/daniellustig/pipecheck`, 2014.

[Lus15] Daniel Lustig. ArMOR: Defending against consistency model mismatches in heterogeneous architectures. `https://github.com/daniellustig/armor`, 2015.

[Mar05] Milo M. K. Martin. Formal verification and its impact on the snooping versus directory protocol debate. *33rd International Conference on Computer Design (ICCD)*, 2005.

[McM01] Kenneth L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Correct hardware design and verification methods*, pages 179–195. Springer, 2001.

[MHAM10] Sela Mador-Haim, Rajeev Alur, and Milo MK Martin. Generating litmus tests for contrasting memory consistency models. In *Computer Aided Verification*, pages 273–287. Springer, 2010.

[MHAM11] Sela Mador-Haim, Rajeev Alur, and Milo MK Martin. Litmus tests for comparing memory consistency models: how long do they need to be? In *48th Design Automation Conference (DAC)*, 2011.

[MHC+06] Chaiyasit Manovit, Sudheendra Hangal, Hassan Chafi, Austen McDonald, Christos Kozyrakis, and Kunle Olukotun. Testing implementations of transactional memory. In *15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006.

[MHMS+12] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. *24th International Conference on Computer Aided Verification (CAV)*, 2012.

[MHW03] Milo MK Martin, Mark D. Hill, and David Wood. Token coherence: Decoupling performance and correctness. In *30th International Symposium on Computer Architecture (ISCA)*, 2003.

[MLPM15] Yatin Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Ccicheck: Using μhb graphs to verify the coherence-consistency interface. *48th International Symposium on Microarchitecture (MICRO)*, 2015.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. *38th Design Automation Conference (DAC)*, 2001.

[Moo65] Gordon Moore. Cramming more components onto integrated circuits. *Electronics*, 38:114–117, April 1965.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. *32nd Symposium on Principles of Programming Languages (POPL)*, 2005.

[MRP+14] Paul E. McKenney, Torvald Riegel, Jeff Preshing, Hans Boehm, Clark Nelson, and Olivier Giroux. Towards implementation and use of `memory_order_consume`. *ISO SC22 WG21 N4321*, November 2014.

[MS05] Albert Meixner and Daniel J. Sorin. Dynamic verification of sequential consistency. In *32nd International Symposium on Computer Architecture (ISCA)*, 2005.

[ND13] Brian Norris and Brian Demsky. CDSchecker: Checking concurrent data structures written with c/c++ atomics. *28th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.

[NSS+09] Francesco Zappa Nardelli, Peter Sewell, Jaroslav Sevcik, Susmit Sarkar, Scott Owens, Luc Maranget, Mark Batty, and Jade Alglave. Relaxed memory models must be rigorous. *2nd International Workshop on Exploiting Concurrency Efficiently and Correctly (EC)^2, in conjunction with CAV*, 2009.

[NVIa] NVIDIA. Kepler tuning guide. `http://docs.nvidia.com/cuda/kepler-tuning-guide`.

[NVIb] NVIDIA. NVIDIA Tegra K1: A new era in mobile computing. `http://www.nvidia.com/content/pdf/tegra_white_papers/tegra_k1_whitepaper_v1.0.pdf`, 2014.

[NVI13a] NVIDIA. CUDA C programming guide v5.5, 2013.

[NVI13b] NVIDIA. Parallel thread execution ISA, v3.2, 2013.

[NVI15] NVIDIA. CUDA toolkit documentation. `http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-ref`, 2015.

[OCY+15] Marc S. Orr, Shuai Che, Ayse Yilmazer, Bradford M. Beckmann, Mark D. Hill, and David A. Wood. Synchronization using remote-scope promotion. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[OSS09a] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2009.

[OSS09b] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, University of Cambridge, Computer Laboratory, March 2009.

[PCC+14] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *41st International Symposium on Computer Architecture (ISCA)*, 2014.

[PCW14] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. *41st International Symposium on Computer Architecture (ISCA)*, 2014.

[PPA+13a] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. Efficient spatial processing element control via triggered instructions. *IEEE MICRO*, 34, May-June 2013.

[PPA+13b] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. Triggered instructions: a control paradigm for spatially-programmed architectures. *40th International Symposium on Computer Architecture (ISCA)*, 2013.

[PRZ01] Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. Automatic deductive verification with invisible invariants. In *7th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2001.

[PSCH98] Manoj Plakal, Daniel J. Sorin, Anne E. Condon, and Mark D. Hill. Lamport clocks: verifying a directory cache-coherence protocol. In *10th Symposium on Parallel Algorithms and Architectures (SPAA)*, 1998.

[PT87] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[PVJ15] Gustavo Petri, Jan Vitek, and Suresh Jagannathan. Cooking the books: Formalizing JMM implementation recipes. *29th European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[Qad03] Shaz Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. *IEEE Transactions on Parallel and Distributed Systems*, 14(8):730–741, 2003.

[Qem15] QEMU. http://wiki.qemu.org, 2015.

[Qua15] Qualcomm. Qualcomm snapdragon 810 processor. 2015.

[RK12] Alberto Ros and Stefanos Kaxiras. Complexity-effective multicore coherence. *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[RLS10] Bogdan F. Romanescu, Alvin R. Lebeck, and Daniel J. Sorin. Specifying and dynamically verifying address translation-aware memory consistency. *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[RZFH06] Amitabha Roy, Stephan Zeisset, Charles J. Fleckenstein, and John C. Huang. Fast and generalized polynomial time memory consistency verification. *18th International Conference on Computer Aided Verification (CAV)*, 2006.

[S+] Peter Sewell et al. C/C++11 mappings to processors. http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.

[ŠA08] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. *European Conference on Object Oriented Programming (ECOOP)*, 2008.

[SD87] Christoph Scheurich and Michel Dubois. Correct memory operations of cache-based multiprocessors. *14th International Symposium on Computer Architecture (ISCA)*, 1987.

[SFW+05] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent Java programs. *10th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[Shi] Anand Lal Shimpi. AMD announced K12 core: Custom 64-bit ARM design in 2016. http://www.anandtech.com/show/7990/amd-announces-k12-core-custom-64bit-arm-design-in-2016.

[SHW11] Daniel Sorin, Mark Hill, and David Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[SKA13] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. DeNovoND: efficient hardware support for disciplined non-determinism. *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[SKM15] Magnus Själander, Stefanos Kaxiras, and Margaret Martonosi. *Power-Efficient Computer Architectures: Recent Advances*. Number 30 in Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2015.

[SMO+12] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and Power. *33rd Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[SN04] Robert C. Steinke and Gary J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM (JACM)*, 51(5):800–849, 2004.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. *21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.

[SNM+12] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. End-to-end sequential consistency. *39th International Symposium on Computer Architecture (ISCA)*, 2012.

[SOIG07] Konrad Slind, Scott Owens, Juliano Iyoda, and Mike Gordon. Proof producing synthesis of arithmetic and cryptographic hardware. *Formal Aspects of Computing*, 19(3):343–362, 2007.

[SPA94a] SPARC. SPARC architecture manual, version 8. 1994.

[SPA94b] SPARC. SPARC architecture manual, version 9. 1994.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(2):282–312, 1988.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. *32nd Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[SSH+13] Samantika Subramaniam, Simon C. Steely, Will Hasenplaugh, Aamer Jaleel, Carl Beckmann, Tryggve Fossum, and Joel Emer. Using in-flight chains to build a scalable cache coherence protocol. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):28, 2013.

[STM14] Divjyot Sethi, Muralidhar Talupur, and Sharad Malik. Using flow specifications of parameterized cache coherence protocols for verifying deadlock freedom. In *12th International Symposium on Automated Technology for Verification and Analysis (ATVA)*. 2014.

[Sun07] Sun. OpenSPARC T2 core microarchitecture specification, rev. A, Dec. 2007.

[ŠVZN+13] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *Journal of the ACM (JACM)*, 60(3):22, 2013.

[TDF+02] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, January 2002.

[Tex10] Texas Instruments. Omap4430 multimedia device technical reference manual, silicon revision 2.0, version j. August 2010.

[The04] The Coq development team. *The Coq proof assistant reference manual, version 8.0*. LogiCal Project, 2004.

[Top15] Top500. `http://www.top500.org`, June 2015.

[TQB+98] Scott Taylor, Michael Quinn, Darren Brown, Nathan Dohm, Scot Hildebrandt, James Huggins, and Carl Ramey. Functional verification of a multiple-issue, out-of-order, superscalar Alpha processor-the DEC Alpha 21264 microprocessor. *35th Design Automation Conference (DAC)*, 1998.

[Tur37] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42:230–265, 1937.

[TVD10] Emina Torlak, Mandana Vaziri, and Julian Dolby. MemSAT: Checking axiomatic specifications of memory models. *31st Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[VBC+15] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *42nd Symposium on Principles of Programming Languages (POPL)*, pages 209–220, 2015.

[VCAD15] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular deductive verification of multiprocessor hardware designs. *27th International Conference on Computer Aided Verification (CAV)*, 2015.

[VN11] Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying fence elimination optimisations. *20th International Symposium on Static Analysis (SAS)*, 2011.

[VT14] Ashish Venkat and Dean M. Tullsen. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. *41th International Symposium on Computer Architecture (ISCA)*, 2014.

[VV14] Gwendolyn Voskuilen and TN Vijaykumar. Fractal++: Closing the performance gap between fractal and conventional coherence. In *41st International Symposium on Computer Architecture (ISCA)*, 2014.

[WBDB15] John Wickerson, Mark Batty, Alastair F. Donaldson, and Bradford M. Beckmann. Remote-scope promotion: clarified, rectified, and verified. *30th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2015.

[YGLS03] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *Correct Hardware Design and Verification Methods*, volume 2860. 2003.

[YGLS04] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. *18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[ZBES14] Meng Zhang, J.D. Bingham, J. Erickson, and D.J. Sorin. PVCoherence: Designing flat coherence protocols for scalable verification. In *20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[ZLS10] Meng Zhang, Alvin R. Lebeck, and Daniel J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *43rd International Symposium on Microarchitecture (MICRO)*, 2010.