Progressive Automated Formal Verification of Memory Consistency in Parallel Processors

Yatin Avdhut Manerkar

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
Computer Science
Adviser: Professor Margaret Martonosi

January 2021

Abstract

In recent years, single-threaded hardware performance has stagnated due to transistor-level limitations stemming from the end of Moore's Law and Dennard scaling. Instead, today's designs improve performance through *heterogeneous parallelism*: the use of multiple distinct processing elements on a chip, many of which are specialised to run specific workloads. The processing elements in such architectures often communicate and synchronise with each other via loads and stores to shared memory. *Memory consistency models* (MCMs) specify the ordering rules for such loads and stores. MCM verification is thus critical to parallel system correctness, but is notoriously hard to conduct and requires examining a vast number of scenarios.

Verification using formal methods can provide strong correctness guarantees based on mathematical proofs, and is an excellent fit for MCM verification. This dissertation makes several contributions to automated formal hardware MCM verification, bringing such techniques much closer to being able to handle real-world architectures. Firstly, my RTLCheck work enables the automatic linkage of formal models of design orderings to RTL processor implementations. This linkage helps push the correctness guarantees of design-time formal verification down to taped-out chips. The linkage doubles as a method for verifying microarchitectural model soundness against RTL. Secondly, my RealityCheck work enables scalable automated formal MCM verification of hardware designs by leveraging their structural modularity. It also facilitates the modular specification of design orderings by the various teams designing a processor. Thirdly, my PipeProof work enables automated all-program hardware MCM verification. A processor must respect its MCM for all possible programs, and PipeProof enables designers to prove such results automatically.

This dissertation also proposes *Progressive Automated Formal Verification*, a novel generic verification flow. Progressive verification emphasises the use of automated formal verification at multiple points in system development—starting at early-stage design—and the linkage of the various verification methods to each other. Progressive verification has multiple benefits, including the earlier detection of bugs, reduced verification overhead, and reduced development time. The combination of PipeProof, RealityCheck, and RTLCheck enables the progressive verification of MCM properties in parallel processors, and serves as a reference point for the development of future progressive verification flows.

Acknowledgements

A dissertation may be the single-authored work of a PhD student, but it is rarely completed due to the efforts of that student alone. In my case, there are many people who have helped me in my journey towards and through grad school, and to whom I offer my sincerest thanks.

First and foremost, I must thank my adviser, Prof. Margaret Martonosi. She gave me an offer of admission when few others did, and has stood by me through both good times and bad. A student's adviser is the most important person in their graduate school travails, and I could not ask for a better adviser than Margaret. I thank her for valuable advice and guidance on a plethora of subjects over the years, and for teaching me how to do research that is truly meaningful. Margaret is an exemplary role model for any young professor to aspire to, and I hope to implement the lessons I learned from her in my future academic career.

I would never have gotten to this point in my life if not for the unconditional love and support of my parents and sister. Whether it was moving to another country with my education in mind, coming to visit me and helping me keep my life in order, giving me a home that was always open to me, or just being a sympathetic ear for the frustrations of grad school, they have *always* been there for me when I needed them. I am eternally grateful for all they have done (and continue to do) for me. I also thank my aunt and her family for being my family in the New York area during my PhD. They were always ready to provide help of a more local nature, and I'm very grateful for all the times that they did.

Dan Lustig has been a valuable mentor and friend to me for my entire time at Princeton. As a senior PhD student, Dan helped me enormously in finding my feet at Princeton and learning how to do research. After he graduated, I benefited from his advice during our weekly teleconferences (which for a while required him to be online at 7:30am). Without him, I may never have looked at memory consistency verification for my PhD research—or gone into formal methods at all. I thank him for showing me that architecture research doesn't have to be about showing 20% speedup on a set of benchmarks, for advice and counsel numerous times over the years, and for being the best damn mentor that any first-year PhD student could ever hope to have. I fondly recall us discussing litmus test μhb graphs on the whiteboard outside our offices in my first year, and I look forward to more such discussions in the future.

Aarti Gupta has been a tremendous positive influence on my dissertation and research trajectory ever since I asked her to serve on my generals committee (on Margaret's advice). I thank her for being my guide to the world of formal verification, for giving me a rigorous foundation of formal methods knowledge on which to build my future research, and for patiently answering all my questions about formal verification, however basic they may have been. Aarti is a veritable ocean of formal verification knowledge. I will never forget the times I asked her what prior work existed on a formal methods topic, and came away with a telling reference that helped me on my way to solving the problem I was facing.

I also thank Margaret, Aarti, Dan, and the other members of my committee, Sharad Malik and Zak Kincaid, for the feedback they have provided on my dissertation work. The questions they raised and the answers I discovered in response have changed this dissertation for the better.

I thank Caroline Trippel—my "research sibling"—for making me laugh, for our many MCM discussions, for teaching me about food and drink, and for valuable job market advice. I must also thank the other current and former members of MRMGroup with whom I interacted over the years: Themis, Tae Jun, Tyler, Aninda, Naorin, Luwa, Prakash, the "Quantum Boyz" Wei and Teague, Marcelo, Yipeng, Esin, Abhishek, Ali, Elba, Logan, Yavuz, Ozlem, and Wenhao. You all are the sauce that makes the meal of grad school palatable, and I am glad to have known each of you. I also thank my other research collaborators, Michael Pellauer and Hongce Zhang, for our productive research discussions.

I'm also indebted to the friends I made at Princeton beyond the confines of our lab. I thank Sumegha Garg and Arjun "Darth" Bhagoji for inviting a lonely grad student to be a part of their friend circles. Through them I made many other friends who I cherish: the Akshays (big and small), Ariel, Nikunj, Divya, Sravya, Gopi, Nivedita, Vivek, Pranav, and Sanjay. I especially thank Arjun and his board game crew for many fun evenings (and for keeping me sane during the pandemic). I thank the PL group—Andrew, Dave, Aarti, Santiago, Qinxiang, Olivier, Joomy, Annie, Ryan, Matt, Nick, Zoe, Lauren, Charlie, Devon, and the rest—for allowing an architect to attend their weekly reading group and for showing me the joys of programming language research. I thank Deep Ghosh for many excellent discussions on *Star Wars* and *The Lord of the Rings*, "Sir" Yogesh Goyal for helpful advice, and both for sharing the joys and sorrows of being a Manchester United fan. And to AB, many thanks for all that you did for me.

I thank the staff at the Davis International Center, especially Katie Sferra, for their assistance in navigating life on a student visa in the US through all the executive orders and presidential proclamations.

I thank Ben Lin for being a stalwart friend through thick and thin for almost 15 years (and hopefully many more), since we were both eager young undergrads at Waterloo. Ben taught me the value of staying positive even in tough times—an essential trait for any grad student.

Finally, while there are many people who helped me through my PhD, there are others who are responsible for me even applying to PhD programs. I thank Tom Wenisch for showing me how much fun research can really be, and for giving me a solid base in computer architecture and parallel computer architecture. I went into my independent work with him looking to get research experience, and emerged with a desire to earn my doctorate. I am very grateful for his support in my decision to come back to academia, and he continues to be someone whose advice I value. I thank Anwar Hasan, Mark Aagaard, and Hiren Patel for awakening my interest in computer architecture at Waterloo. And I thank Mrs. Mitchell, my high school principal, for taking pains to give a young immigrant the best high school education a tiny Canadian town could provide.

To my grandfathers, R.M.S.M. and D.D.N.K.,

both great men in their own ways.

Contents

Chapter 1

Introduction

"Non omnia possumus omnes*, but at least we can step into a boat at a stated time, can we not?"*
—Patrick O'Brian
Post Captain

Computing today has permeated deep into our daily lives. We rely on computers to run our phones, process financial transactions, navigate when travelling, and for many other functions. In other words, computing has become an indispensable tool for humanity to function. This trend will only increase in future years, with the rise of artificial intelligence (AI) and the development of cyber-physical systems like robots and self-driving cars.

The continued importance of computing in our daily lives makes it critical to ensure that these systems run correctly. The ubiquity of computing today means that the ramifications of hardware and software bugs are higher than ever. Computing bugs today can lead to car crashes [Lee19], Internet outages [Str19], and the leakage of confidential information [KHF$^+$19].

At the same time, computing systems have evolved over time to become considerably more complex in the quest for improved performance and energy efficiency. Today's microprocessors are complex integrated circuits consisting of a variety of components working together, and are capable of executing billions of instructions per second [Wal18]. Likewise, today's software is also quite complex, consisting of vast numbers of individual modules connected to each other, and often running on large and distributed collections of nodes. Such software may be comprised of millions of lines of code [Alg17]. These factors make verification of today's computing systems a difficult task.

A key contributor to the complexity of systems today is the fact that they are *parallel* systems. Processors today routinely contain at least 4 to 8 general-purpose processing cores that can operate concurrently, and may contain over 40 accelerators for speeding up specific types of computation [WS19]. On the software side, programming languages today routinely have native support for concurrent threads of computation. These threads and cores process data concurrently and often communicate via shared memory. The *memory consistency model* (MCM) of such a system specifies the ordering rules for the memory operations that are used for such communication. Consequently, if parallel systems do not obey their MCMs, then they are liable to malfunction. This makes verification of MCM implementations critical to parallel system correctness.

The work contained in this dissertation makes significant advances towards thorough, comprehensive verification of MCM properties in both parallel hardware. This chapter provides an overview of the challenges in MCM verification and the solutions proposed by this dissertation to meet those challenges. I begin by examining the trends in systems that have led to MCMs becoming a critical part of most computing systems today.

1.1 The Rise of Heterogeneous Parallelism

1.1.1 The Quest for Improved Hardware Performance

For decades, the interface that hardware provides to the programmer has been that of an instruction-level abstraction (ISA), where (from the point of view of the programmer) processors execute instructions one at a time and in program order. Beneath this abstraction, computer architects and VLSI engineers improved hardware performance along two major axes. The first avenue for performance improvement was at the circuit level, and is often characterised using a combination of Moore's Law [Moo06a] and Dennard Scaling [DGY+74]. Moore's Law predicted that the number of transistors on an integrated circuit would double every two years[1], giving computer architects more transistors to use for their chip. Meanwhile, under Dennard scaling, reducing the dimensions of transistors by a factor of S increased their maximum switching speed (i.e. the maximum clock frequency of the processor) by a factor of S, but their power density could be kept constant by correspondingly reducing their supply voltage by a factor of S. Roughly speaking, Dennard scaling enabled engineers to reduce a chip to half its size and twice its speed at every new generation of transistors, while consuming the same amount of power.

The second avenue for the improvement of performance was through the development of improved hardware designs or *microarchitectures* (as opposed to circuits). Over the years, numerous microarchitectural innovations and optimizations have been developed. For instance, pipelining broke up the execution of an instruction into a number of pieces, and allowed a processor to execute parts of multiple different instructions in the same clock cycle. Since the work performed in each cycle was reduced, the design could then be run at a higher clock frequency, improving performance. Another

[1]Moore initially predicted a doubling of transistors every year, but a decade later revised his estimate to a doubling of transistors every two years [Moo06b].

example is that of CPU caches, which were developed to improve memory latency and reduce the amount of time processors spend waiting on memory [HP17].

Over the years, computer architects also innovated to improve performance through instruction-level parallelism (ILP) [HP17], or the execution of multiple processor instructions in parallel (as opposed to one at a time). ILP includes pipelining, but also a number of other techniques. These techniques include the execution of instructions out of order (when it would not change the results of the computation), prediction of the results of branch instructions before their inputs were ready, and speculative execution of instructions based on the prediction of their inputs or the results of branch prediction. Another technique closely related to ILP is that of memory-level parallelism (MLP), where multiple memory operations are sent to memory in an overlapping or parallel fashion to reduce the overall number of cycles spent waiting on memory [Gle98]. Notably, each of these schemes took care to ensure that their use did not change the results of the program (which was a single stream of instructions).

1.1.2 The Multicore Era and Heterogeneity

Figure 1.1[2] [Rup20] shows trends in microprocessor data over the last 48 years. It does so by plotting attributes (including processor clock frequency, number of cores, and power usage) of various real processors over that time period. As Figure 1.1 shows, around the mid-2000s, Dennard scaling began to break down, primarily due to leakage power issues. The supply voltage of a transistor (V_{DD}) needs to stay above its threshold voltage (V_t) in order for the transistor to function as a binary switch. Thus, decreasing V_{DD} in keeping with Dennard scaling requires decreasing V_t as well. However, leakage power (power consumed by a transistor regardless of whether it is switching) grows exponentially as threshold voltage decreases [Bos11b]. This makes

[2]Figure 1.1 is "48 Years of Microprocessor Trend Data" by Karl Rupp (https://www.karlrupp.net/), and is licensed under the 'Creative Commons Attribution 4.0 International Public License' (https://creativecommons.org/licenses/by/4.0/legalcode).

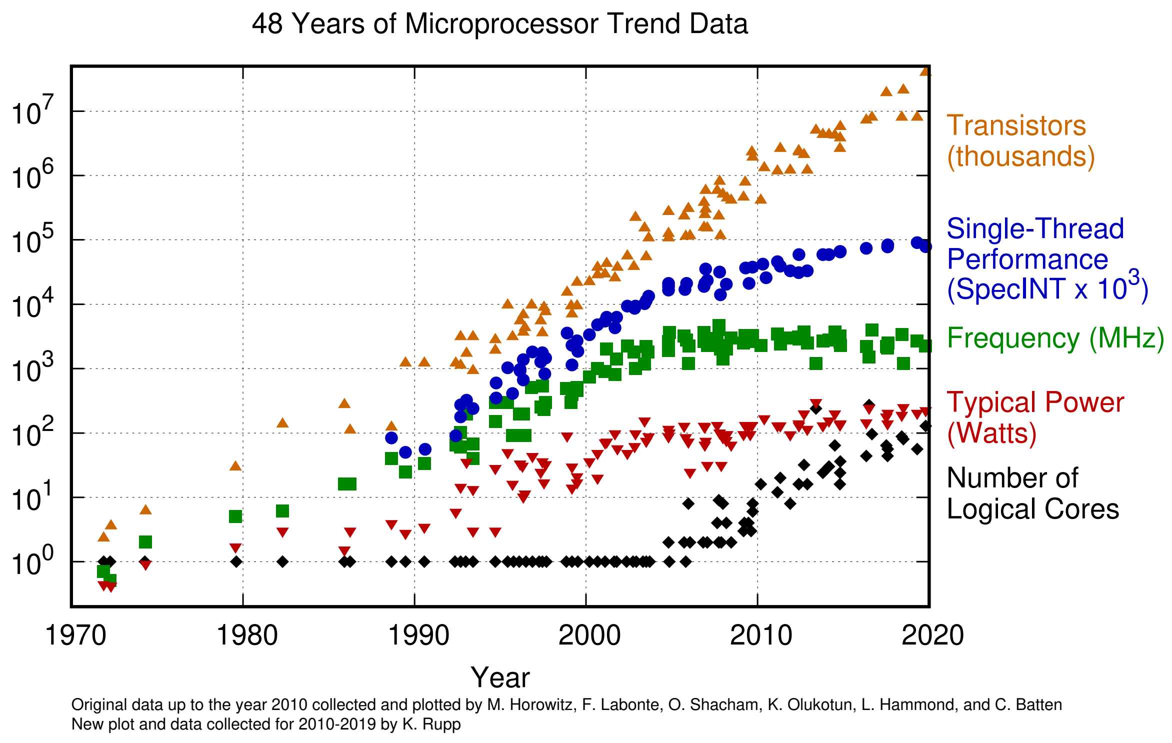


Figure 1.1: Trends in microprocessor parameters, including clock frequency, power usage, and number of cores over the last 48 years [Rup20].

it impossible to keep power density constant when decreasing transistor dimensions beyond a certain point, leading to the breakdown of Dennard scaling.

While architects and circuit engineers were unable to keep power density constant, they were still able to decrease transistor dimensions and increase the number of transistors on a single chip as per Moore's Law. However, due to the breakdown of Dennard scaling, these devices could not turn all their transistors on at the same time, as the combined power usage of all the transistors would be too high.[3] The issues related to power delivery and dissipation encountered by architects and circuit designers when trying to scale designs beyond the realms of Dennard scaling came to be known as the "power wall" [Bos11b].

Faced with the power wall and the end of Dennard scaling, architects could no longer increase processor clock frequency in order to improve performance. Instead,

[3]In recent years, this inability to activate all transistors at once has become known as the "dark silicon" phenomenon [GSV+10].

they began using the extra transistors on chips to create multiple processing *cores*, each of which could execute its own stream of instructions in parallel. Figure 1.1 depicts this through the increase in the number of logical cores per chip from the mid-2000s onwards. Theoretically, an N-core version of a processor could provide N times the performance of a single-core version of that processor, due to its ability to execute N instruction streams in parallel. This would provide large speedups without having to increase clock frequency (and thus power usage). However, the extent to which such a multicore processor can be utilised is dependent on how much the application can be parallelised (i.e., split into multiple pieces that can be run in parallel with each other). For instance, a linear speedup (i.e., N times the performance of single-core when using an N-core processor) is only possible for applications that can be parallelised into N equal parts that can each run in parallel with each other.

In recent years, architects have also begun to adopt *heterogeneity* as a means to develop energy-efficient processors. Heterogeneity is the use of specialised cores for specific types of tasks. Specialising cores to execute specific types of workloads allows architects and circuit engineers to optimise for those workloads, and remove unnecessary general-purpose functionality from such accelerators. This results in huge performance improvements and energy efficiency gains for the workloads that the accelerator is designed for. The field of computer architecture has seen a wide variety of accelerators in recent years, including for machine learning [CDS+14] and graph analytics [HWS+16] workloads. In fact, the recent Apple A12 System-on-Chip (SoC) contains more than 40 accelerators [WS19].

This heterogeneous parallelism of today's hardware has percolated up the stack to the level of high-level languages (HLLs) and software. Today's programming languages like C [ISO11a], C++ [ISO11b], and Java [GJS+14] all have native thread support to enable parallel programming. Operating systems such as the Linux kernel have been revised to be concurrent in order to make the best use of today's parallel

Thread 0	Thread 1

(i1) x = 1;
(i2) if (y == 0) {
critical section
}
	(i3) y = 1;
	(i4) if (x == 0) {
	critical section
	}

Can both threads enter their critical section at the same time?	

Figure 1.2: A simple synchronisation algorithm based on Dekker's algorithm [Dij02]. This program can be used to illustrate how counterintuitive program outcomes can occur due to MCM issues. The initial values of `x` and `y` are assumed to be 0.

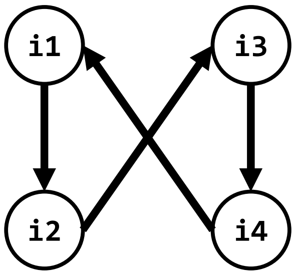


Figure 1.3: The happens-before orderings for Figure 1.2's program that are required for both loads (`i2` and `i4`) to return 0 under SC. The cycle in the graph indicates that such a result is impossible under SC (as it would require an instruction to happen before itself).

hardware [AMM+18]. Furthermore, with the advent of heterogeneity, specialised toolchains have been developed for certain types of accelerators. TensorFlow [Goo20] and PyTorch [F+20] are two such flows for machine learning accelerators. The programming of systems that support shared-memory concurrency necessitates memory consistency models to describe the behaviour of the parallel system's shared memory, as discussed next.

1.2 The Need for Memory Consistency Models

The various cores in a parallel processor often interact with each other via a shared memory abstraction. Cores thus synchronise and communicate with each other through load and store operations to shared memory. In a single-core system with a single instruction stream, figuring out the result of each load instruction is quite

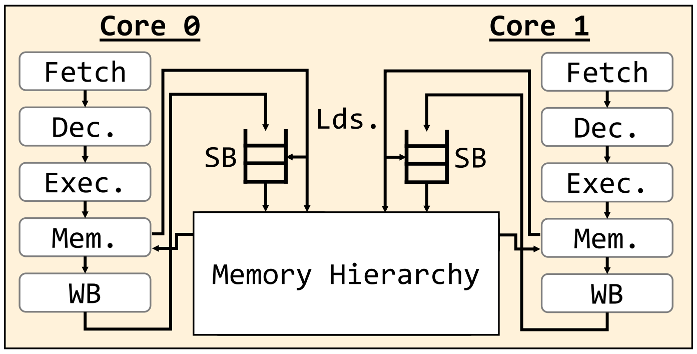


Figure 1.4: A microarchitecture with two cores, each with five-stage pipelines and a store buffer (SB). This design is a simplistic example of how common hardware optimizations like store buffers can lead to weak MCM behaviours.

straightforward. If we define *program order* as the order in which instructions appear in the program's instruction stream, then a load to a given address `x` simply returns the value written by the last store to address `x` before that load in program order. However, in a multicore processor, deducing the values that a load can return is substantially more involved.

Consider the simple synchronisation algorithm in Figure 1.2, based on Dekker's algorithm [Dij02]. Here, two threads each want to access the same shared resource in their critical section. Assume all addresses have a value of 0 before the execution of the program. Each thread has one flag variable (which is shared with the other thread): thread 0's flag is `x` and thread 1's flag is `y`. In this synchronisation algorithm, each thread sets its own flag and then reads the value of the other thread's flag. A thread only enters its critical section if its flag is set and the other thread's flag is not set. Can the loads in Figure 1.2's program return values other than 1 or 0? Is it valid for both loads (`i2` and `i4`) to return a value of 0 in the same execution? These questions necessitate a specification for the behaviour of the parallel processor's shared memory abstraction, called a *memory consistency model* or *memory model* [AG96], henceforth abbreviated as MCM.

MCMs specify the ordering rules for memory and synchronization operations in parallel programs. In essence, they dictate what value will be returned when a parallel program does a load. Since the multicore era, the ISA specifications of parallel processors have been updated to include MCM specifications for processors implementing those ISAs. The simplest and most intuitive MCM is *sequential consistency* (SC), defined by Leslie Lamport in 1979 [Lam79]. Under sequential consistency, the results of the overall program must correspond to some in-order interleaving of the program statements or instructions from each individual core or thread. Each core or thread must execute its memory operations in program order. Only one thread or core can execute memory operations at any given time. Furthermore, each memory operation must be performed atomically—in other words, it must become visible to all threads or cores at the same time. Each load must return the value of the last store to its address in the overall total order of memory operations corresponding to its execution.

To illustrate SC, consider the execution of Figure 1.2's program. Under SC, can both the loads `i2` and `i4` return 0 in the same execution? For `i2` to return 0, it must execute before `i3`. Likewise, for `i4` to return 0, it must execute before `i1`. However, since each thread must execute its memory operations in program order, `i1` must execute before `i2` and `i3` before `i4`. Each of these orderings is reflected by an arrow in Figure 1.3's graph. Together, they form a cycle, indicating that for both `i2` and `i4` to return 0, one or more of the program statements must happen before itself, which is impossible. Thus, under SC, it is forbidden for both `i2` and `i4` to return 0 in the same execution. The synchronisation algorithm will therefore work as expected under SC, and at most one thread will be in its critical section at any time.

While sequential consistency is a simple and intuitive MCM, its semantics put it at odds with the vast majority of processor microarchitectures today. The microarchitectural optimizations mentioned in Section 1.1.1 were developed in the single-core era, where processors only had one core and one instruction stream operating at any time. Microarchitectural features like store buffers, out-of-order execution, and memory-level parallelism (MLP) routinely reorder the execution of memory instructions to improve performance, but such reordering violates SC. Consider the execution of Figure 1.2's program on the microarchitecture in Figure 1.4, where each core has a five-stage pipeline and its own store buffer [HP17]. (Assume thread 0 runs on core 0 and thread 1 runs on core 1.) Since memory latencies are high, cache misses on stores[4] can notably increase execution time. In this microarchitecture, instead of sending stores directly to the memory hierarchy (and stalling the pipeline until they complete), cores send stores to the store buffer and continue execution of other instructions while waiting for the stores to complete. The store buffer handles sending stores to the memory hierarchy independently of the core's pipeline. The store buffer thus allows the core to overlap the latency of store misses with the execution of other instructions, thus reducing the effective latency of store misses. Subsequent loads check the store buffer for their address to ensure they get the latest value. Loads return the value of the latest entry for their address from the store buffer if one exists, and go to the memory hierarchy otherwise.

While the use of store buffers improves performance, it also makes it possible for both loads (`i2` and `i4`) in Figure 1.2's program to return 0. In particular, store buffers allow core 0 to put `i1` in its store buffer and send `i2` to memory before `i1` has completed, thus reordering `i1` and `i2` from the point of view of other cores. If we consider thread 0's part of Figure 1.2's program in isolation, then the use of a store buffer would not change the result. The final value of `x` would be 1, and the final value of `y` would still be 0. However, in a multicore context, the reordering becomes architecturally visible (i.e., reflected in the values returned by the program's loads). Specifically, core 0 could put `i1` in its store buffer, and core 1 could do the same with `i3`. Then, core 0 and core 1 could send `i2` and `i4` to memory before the stores `i1` and `i3` completed, thus allowing both loads to return 0. In a nutshell, the microarchitectural optimizations that were invisible to programmers in a single-core

[4]Assuming a write-allocate cache.

setting become programmer-observable when those cores are used in a multicore processor.

In order to implement SC, architects would either have to forgo any optimizations that reordered memory operations or only reorder memory operations speculatively [Hil98, BMW09]. (Section 2.1.2 provides further details on the latter.) Forgoing optimizations that reorder memory operations would result in a huge performance hit, which is unacceptable to the majority of processor users. Meanwhile, speculative reordering of memory operations results in complex designs that must keep track of when reorderings are observed by other cores and roll back appropriately. Furthermore, speculative execution is vulnerable to side-channel attacks [KHF$^+$19, TLM18b].

As a result, most of today's processors (including all commercial ones) do not implement SC. Instead, they implement *weak* or *relaxed* MCMs (also known as *weak memory models*) [OSS09, PFD$^+$18, AMT14, RIS19] that relax one or more types of orderings among memory operations.[5] For instance, the MCM of x86 processors from Intel and AMD is Total Store Order (TSO) [OSS09], which relaxes orderings between stores and subsequent loads. Thus, unlike SC, TSO allows the outcome of Figure 1.2's program where both loads return 0. To enable synchronization between cores, ISAs that implement weak MCMs provide other instructions capable of enforcing ordering between memory operations where necessary. For example, x86 provides an `MFENCE` (memory fence) instruction that can be placed between `i1` and `i2` and between `i3` and `i4` in Figure 1.2 to prevent the hardware from reordering those two stores. A possible implementation of the `MFENCE` would be to drain the store buffer before executing any memory operations that are after the `MFENCE` in program order. (Section 2.1.3 provides further background information on weak MCMs.)

[5]Processors implementing weak MCMs may also speculatively reorder memory operations other than those relaxed by their MCM, and roll back such speculation if the speculative reorderings would be detected.

Most programmers do not program in the assembly language of a processor's ISA, but in high-level programming languages like C, C++, and Java. These languages have their own MCMs [ISO11a, ISO11b, GJS+14] which describe the behaviour of memory as seen by the various threads in a high-level language program. Most programming language MCMs are based on the SC-for-DRF theorem [AH90, GLL+90], which enables the program to behave as if the hardware were implementing SC as long as the programmer has added sufficient synchronization between the threads in their program. Section 2.1.3 contains a brief overview of the SC-for-DRF theorem.

1.3 The Need for MCM Verification

MCMs are defined at interfaces between layers of the hardware-software stack. The MCM of such an interface describes the ordering guarantees that must be provided by lower layers of the stack, and consequently the ordering guarantees that upper layers of the stack can rely on. For example, the MCM of an instruction set like x86 defines the ordering guarantees that must be provided by x86 hardware for any parallel program. Likewise, it defines the ordering guarantees that any x86 assembly-language program can expect the hardware to maintain. Similarly, the MCM of a high-level language like C++ or Java defines the ordering guarantees that a programmer can expect the language runtime to provide for their code. These required orderings must be maintained by the lower layers of the stack—in this case, a combination of the compiler and hardware.

If the layers of a parallel system do not obey their MCMs, then parallel programs will not run correctly on the system. As an example, consider once more the program from Figure 1.2. A programmer might write such code for a processor whose MCM is SC, and use it to synchronise between threads. In such a case, the programmer would expect that at most one thread would be in its critical section at any time (i.e., both loads could never return 0). However, if the processor was buggy and violated SC—for instance, by using store buffers to conduct non-speculative reordering of memory operations—this would no longer be true. It would then be possible for both threads to enter their critical section at the same time, thus breaking the synchronisation mechanism. As such, MCM verification is critical to parallel system correctness.

With concurrent systems becoming both more prevalent and more complex, MCM-related bugs involving hardware are more common than ever. Intel processors have experienced at least two transactional memory bugs in recent years [Hac14, Int20]. In another case, ambiguity about the ARM ISA-level MCM specification led to a bug where neither hardware nor software was enforcing certain orderings [AMT14, ARM11]. With no cost-effective hardware fix available, ARM had to resort to a low-performance compiler fix instead. MCM-related issues have also surfaced in concurrent GPU programs [ABD$^+$15, SD16]. Finally, the computer security community has recognized that MCM bugs may lend themselves to security exploits [GNBD16]. The work in this dissertation also uncovers an MCM bug in an existing system (see Section 3.9.1). This illustrates that MCM bugs will continue to occur without careful design and the use of verification approaches such as those in this dissertation.

The traditional method of conducting verification is to run tests on the system, but such dynamic verification approaches are insufficient for conducting MCM verification. The reason for this is that today's parallel systems (including multicore processors) are nondeterministic [DLCO09]. This means that a parallel program may give one result when run once on a multicore processor, and a different result when run a second time. For instance, even on a processor implementing SC, Figure 1.2 may give one of 3 results: `i2` may return 1 and `i4` may return 0, `i2` may return 0 while `i4` returns 1, or both loads may return 1. Which execution occurs depends on the order in which the stores and loads reach memory from the two cores. Furthermore, even two executions that give the same architectural result may differ microarchitecturally. For example, in one execution a load may read a value from the cache (if it was already present there), and in another execution the load might fetch the same value from main memory. As a result, even if a program such as Figure 1.2 were run a million times on a processor and the outcome where both loads return 0 never showed up, this does not guarantee that the outcome will never occur on the processor.

If using dynamic verification, there is no way to guarantee that one has tested all possible microarchitectural executions for a given program. Instead, MCM properties are best suited to be verified using *formal methods*. In formal verification, a *model* (i.e., a representation) of the hardware or software being verified is specified in some form of mathematical logic. The verification procedure then uses mathematical techniques to formally prove the correctness of the model. Provided that the model accurately represents the system being verified, a proof of the model's correctness implies that the hardware or software is in fact correct. For the rest of this dissertation, unless otherwise specified, I use the term "verification" to mean verification using formal methods. Meanwhile, I use the term "validation" to mean all techniques for checking or ensuring the correctness of a system, including both dynamic testing-based methods and formal verification.

Formal verification approaches can vary in the degree of automation they support. One one end, some formal verification methods (e.g., Kami [VCAD15,CVS+17]) require users to manually write proofs of theorems in a proof assistant, and an automated tool then checks that the proofs do indeed prove the corresponding theorems. On the other hand, formal verification approaches like *model checking* [CHV18] (Section 2.2.1) generally only require users to write a specification of the system and the property they wish to verify. An automated tool then either proves the property correct or returns a *counterexample* (an execution that does not satisfy the property). Other verification approaches may be partially automated, e.g., Ivy [PMP+16] conducts interactive verification guided by the user. The work in this dissertation utilises automated formal verification approaches based on model checking. Section 2.2 provides further background on formal verification.

Since MCMs are defined for both processor ISAs and high-level programming languages, MCM verification is a full-stack problem.[6] Hardware engineers need to verify that their processor correctly implements its MCM, but writers of compilers for high-level languages must also ensure that the code generated by their compilers maintains the ordering guarantees of the high-level language MCM. If either the hardware or compiler do not maintain their required MCM guarantees, parallel programs will not run correctly on the hardware. Furthermore, if the hardware and software do not agree on who is responsible for maintaining which part of high-level language MCM guarantees, this can result in incorrect program executions as well [ARM11].

In this dissertation, the term "MCM verification" refers to verification that hardware and the software runtime of a high-level language (e.g. compiler and OS) correctly maintain the MCM guarantees of the hardware ISA and the high-level programming language. In contrast, this dissertation does not focus on program verification under various MCMs [ND13, OD17, AAA+15, AAJL16, AAJ+19, KLSV17, KRV19], which assumes that the hardware and software runtime correctly maintain MCM guarantees and verifies that a program satisfies its high-level specification (i.e., the programmer's intent). (Section 2.3.2 surveys prior work on program verification under various MCMs.)

In addition to being a factor across the hardware/software stack, MCMs have ramifications throughout the hardware design timeline. While verification is traditionally thought of as something that is conducted post-implementation, one can derive large benefits by beginning hardware MCM verification much earlier. At the point of early-stage processor design, the choice of MCM for a processor can enable or

[6]MCMs are sometimes also defined for intermediate representations, like the LLVM IR [CV17] or NVIDIA PTX [LSG19].

preclude the use of certain microarchitectural features. For instance, choosing SC as a processor's MCM prevents the use of store buffers (except through speculation and rollback; see Section 2.1.2). If a processor design utilises microarchitectural features (or a combination thereof) that violate its MCM, but this design flaw is only caught in post-implementation verification, then this necessitates a redesign (significantly delaying the development of the chip) or a software workaround [ARM11] (which can be heavy-handed and reduce performance). It would be much more efficient to catch such design bugs as early as possible by conducting design-time MCM verification during early-stage design.

As the design evolves, MCM verification should continue to be conducted to ensure that the development of the design does not introduce any bugs that would violate the MCM. Of course, once the design is implemented in Register Transfer Language (RTL) like Verilog, the implementation must also be verified to ensure that it meets MCM requirements.

1.4 Unresolved Challenges in MCM Verification

In response to the need for MCM verification, there has been a large amount of work on formally specifying and verifying MCMs in the past decade or so. (Chapter 2 provides a survey of this related work.) A large portion of prior formal MCM specification and verification work treated hardware as correct (i.e. matching architectural intent) and/or only verified the MCM correctness of hardware using dynamic testing [OSS09,AMSS11, SSA+11,MHMS+12,AMT14,ABD+15,GKM+15,FGP+16,WBBD15,WBSC17,FSP+17, AMM+18,CSW18,PFD+18]. As Section 1.3 above explains, dynamic testing cannot guarantee the absence of bugs (even for tested programs) due to the nondeterminism of today's multiprocessors. As such, there existed a gap in the MCM verification of parallel systems that needed to be filled through the development of formal MCM verification approaches for hardware designs and implementations.

In recent years, there have been two lines of work for the formal MCM verification of hardware. One approach, Kami [VCAD15, CVS+17], enables designers to write their hardware in a proof assistant and manually prove its correctness. A proof assistant like Coq [Coq04] is a tool that aids users in writing proofs of theorems that they can specify in the proof assistant. The proof assistant is capable of verifying whether the user's proof does indeed prove the theorem it purports to, using an approach based on programming language type checking [Chl13]. It also keeps track of what techniques have been used in the proof and what remains to be proven. While Kami is capable of proving the MCM correctness of hardware designs, the proofs require a large amount of manual effort and formal methods expertise. Thus, to produce an efficient formally verified hardware design using Kami, one would need expertise in both computer architecture and formal methods. However, most computer architects do not have formal methods expertise (and vice versa). As such, Kami is not well-suited for use by typical computer architects. (Section 2.3.4 provides further background on Kami.)

The other prior approach for formal MCM verification of hardware designs is PipeCheck [LPM14, LSMB16], which is based on model checking [CHV18]. (Sections 2.2.1 and 2.4 provide further background on model checking and PipeCheck respectively.) PipeCheck was the first automated approach for MCM verification of hardware designs against ISA-level MCM specifications. Given a microarchitectural ordering specification for a hardware design and a litmus test[7] outcome, PipeCheck is capable of automatically verifying whether the given outcome will ever be observable on the design. PipeCheck verifies one test program at a time, and its automation enables it to be used by computer architects rather than just formal methods experts.

[7]A small 4-8 instruction program used to test or verify MCM implementations. Section 2.1.1 covers litmus tests in detail.

In this dissertation, I use the term "PipeCheck" to refer to the current incarnation of the PipeCheck approach as realised in its follow-on work COATCheck [LSMB16]. COATCheck built on the initial PipeCheck work [LPM14] by adding a domain-specific language (μspec) for specifying microarchitectural orderings and developing a custom SMT solver to conduct PipeCheck's model checking. COATCheck also added the ability to specify and verify microarchitectural orderings related to virtual memory, but this capability is largely orthogonal to the advances made in this dissertation. Section 2.4 provides further details on the PipeCheck approach.

Even with the creation of PipeCheck, there still remained a number of unsolved research challenges in MCM verification. One such challenge is that of verifying model *soundness*. If a model is sound with respect to a system, then a proof of the model's correctness will imply that the real system is in fact correct. Using an unsound model can result in false negatives (i.e., the model can be formally verified as correct, but the real system is incorrect) which allow bugs to slip through formal verification. Despite the importance of verifying model soundness, prior work either provided no method for soundness verification or only verified model soundness using dynamic testing. For example, if creating a PipeCheck model for the verification of an existing processor, PipeCheck provides no way to formally verify that such a model is sound with respect to the real processor implementation written in RTL like Verilog. Similarly, the formal models of existing ISA-level MCMs (Section 2.3.1) were only verified for soundness against existing implementations of those ISAs through dynamic testing and consultation with architects. In other words, neither ISA-level MCM specifications nor PipeCheck models have been formally verified as being sound with respect to real processor RTL. This increases the possibility of bugs going undetected when such models are used for formal verification.

Another unsolved challenge is that of verification *scalability*. The monolithic MCM verification of approaches like PipeCheck does not scale to large detailed designs such as those used in commercial processors. This is because—like many model checking approaches [BSST09]—PipeCheck uses Satisfiability Modulo Theories (SMT) solvers[8] which are NP-complete [Coo71]. As such, once verification queries are beyond a certain size, the tool's runtime and/or memory usage tends to explode. Approaches like PipeCheck verify the MCM correctness of a design as a single unit, so it is infeasible to use them to verify detailed designs.

A third challenge is that of verification *coverage*. The vast majority of prior automated MCM verification approaches [AMT14, LPM14, LSMB16, WBSC17] only conduct litmus test-based verification or *bounded* verification of a set of test programs.[9] While verifying a processor or compiler for a large number of tests gives engineers confidence that their system is correct, there always remains the possibility that the system contains a bug which can only be detected by a program not present in the test suite. Furthermore, there are an infinite number of possible programs, so they cannot each be tested individually. For hardware MCM verification, engineers had to choose between manual verification across all programs using Kami or automated verification for a bounded set of programs using PipeCheck. No single prior approach provides automated all-program hardware MCM verification.

A fourth challenge deals with the need for MCM verification at multiple points in the development timeline of a system. For example, PipeCheck is intended for MCM verification of early-stage designs, long before they are implemented in RTL. However, MCM verification of RTL implementations is crucial as well, because even if a design is verified as being correct, a buggy implementation of that design in RTL may still violate the processor's MCM. Even so, PipeCheck provides no way to link

[8]Given a propositional logic formula F, a Boolean satisfiability (SAT) solver can examine all possible assignments to the variables in F. If an assignment to the variables exists which makes F true, the solver will return that assignment. Otherwise it will return that the formula is *unsatisfiable*. SMT builds on SAT, adding the ability to express properties and systems using various theories, thus improving both expressibility and verification performance. Section 2.2.1 provides further details on SAT and SMT-based model checking.

[9]Dodds et al. [DBG18] conduct automated MCM verification of C/C++ compiler optimizations across all possible programs, and their work is a notable exception to this rule.

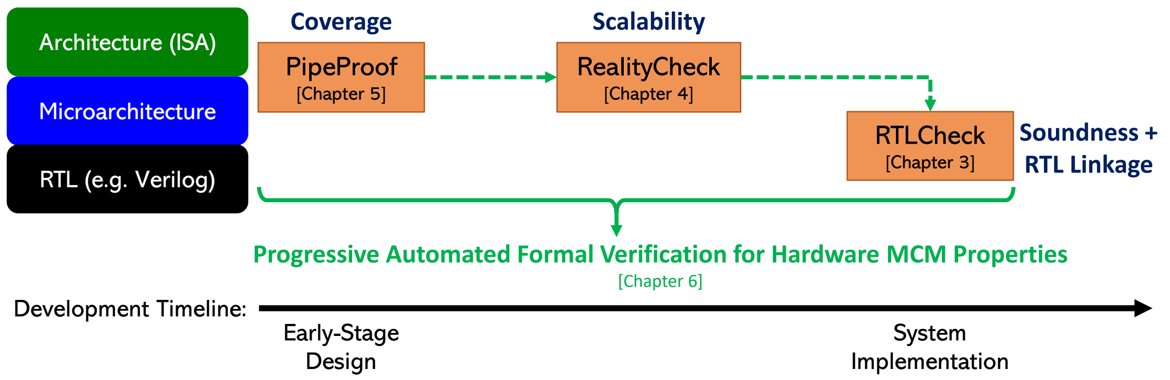


Figure 1.5: The work in this dissertation, organised vertically according to the levels of the hardware/software stack that it covers, and ordered horizontally according to the point in the design timeline it covers. Each work of research is annotated with the challenges in MCM verification that it addresses, as well as the dissertation chapter in which it is covered.

its verification to the verification of the processor's eventual RTL implementations. PipeCheck also provides no easy way for a microarchitectural model to evolve as the design of the processor progresses towards an RTL implementation. There is no straightforward way in PipeCheck to replace the high-level specification of individual components with their detailed design specifications as the design is fleshed out in more detail.

The work in this dissertation develops methodologies and tools to combat these previously unsolved challenges, making large strides forward in the field of MCM verification. These contributions are enumerated next.

1.5 Dissertation Contributions

Figure 1.5 provides a graphical depiction of the work in this dissertation. The work is ordered vertically according to the levels of the hardware/software stack that it covers, and is ordered horizontally according to the point in the design timeline it covers. Figure 1.5 also annotates the parts of the dissertation that address the challenges of soundness, scalability, and coverage respectively.

This dissertation makes the following contributions:

- **Linking Automated MCM Verification to Real Processor Implementations:** This dissertation enables microarchitectural ordering specifications used for automated MCM verification to be linked to RTL implementations written in Verilog for the first time. Prior work on automated hardware MCM verification only went down to microarchitecture and could not be linked to real implementations, limiting its effectiveness in ensuring the correctness of taped-out chips. The linkage developed by my work (RTLCheck) enables correctness guarantees proven for early-stage design ordering specifications to be easily pushed down to the eventual RTL implementations. The linkage of microarchitectural models to RTL also doubles as a mechanism to formally verify the soundness of a microarchitectural model with respect to RTL. Such soundness verification helps engineers develop accurate formal models of existing processor implementations.

- **Scalable Automated Hardware MCM Verification:** This dissertation enables scalable automated MCM verification of detailed hardware designs for the first time. Prior work on automated microarchitectural MCM verification used monolithic approaches that do not scale due to the NP-completeness of the SMT solvers used. However, hardware designs inherently possess large amounts of structural modularity and hierarchy, and my work (RealityCheck) exploits these features to enable specification and verification of a hardware design piece-by-piece. This allows for scalable verification by breaking up a processor's MCM verification into smaller verification problems. In the process, my work also enables modular ordering specifications for hardware designs. Such specifications are an excellent fit for the distributed nature of the hardware design process, where a number of teams each design one or a few components and then connect them together to create the overall processor.

- **Automated All-Program Microarchitectural MCM Verification:** This dissertation is the first to enable automatic proofs of the MCM correctness of hardware designs across all programs. Prior automated microarchitectural MCM verification approaches conducted bounded verification, which only guaranteed design correctness for a subset of all programs. In contrast, my work (PipeProof) develops the first microarchitectural MCM verification approach capable of automatically verifying MCM correctness across all programs. This gives designers complete confidence that there are no MCM bugs in their design, while retaining the ease of verification characteristic of automated approaches.

- **Progressive Automated Formal Verification:** This dissertation proposes *Progressive Automated Formal Verification*, a novel generic verification flow with multiple benefits. Prior formal verification approaches focused on one point in development, like early-stage design or post-implementation verification. In contrast, progressive verification emphasises the use of automated formal verification at multiple points in the development timeline and the linkage of the different verification approaches to each other. Progressive verification enables the earlier detection of bugs and provides reductions in verification overhead and overall development time. In addition, the combination of the PipeProof, RealityCheck, and RTLCheck tools developed by this dissertation enables the progressive verification of MCM properties in parallel processors. This concrete instance of a progressive verification flow serves as a reference point for future work on the progressive verification of other types of properties and systems.

- **Bringing Automated MCM Verification Closer to Real-World Processors:** The work in this dissertation advances automated formal MCM verification much closer to being capable of verifying the designs and implementations of real-world processors. Individually, each of the tools in this dissertation makes its own contribution in this regard. RTLCheck enables automated MCM verification of real processor implementations for the first time. RealityCheck's twin benefits of scalability and distributed specification are both critical to the verification of real-world designs, while PipeProof brings the coverage of automated MCM verification approaches up to the level required for real-world processors. In addition, when the three tools are combined in a progressive verification flow, they enable thorough and efficient MCM verification across much of the hardware development timeline. This thorough progressive verification is essential to ensure the MCM correctness of real-world processors that are shipped to end users.

1.6 Outline

The rest of this dissertation is organised as follows. Chapter 2 provides background information necessary to understand this dissertation, as well as a survey of the relevant related work on formal MCM specification and verification. Chapter 3 details RTLCheck, a methodology and tool for formally checking the soundness of hardware ordering specifications with respect to their RTL implementations. The work in Chapter 3 also doubles as a method to link microarchitectural MCM verification to established RTL verification procedures. Chapter 4 covers RealityCheck, a methodology and tool for achieving scalable automated MCM verification of hardware designs through modularity. Chapter 5 explains PipeProof, a methodology and tool for conducting automated MCM verification of hardware designs across all programs rather than just litmus tests. Chapter 6 details the philosophy of Progressive Automated Formal Verification and its attributes, and illustrates how the work in Chapters 3, 4, and 5 achieves progressive verification of MCM properties for hardware designs. Chapters 3, 4, and 5 contain quantitative evaluations of their methodologies on one or more case studies each. Chapter 7 wraps up the dissertation, providing a brief retrospective on the work presented, outlining avenues for future work, and stating the conclusions of the dissertation.

Chapter 2

Background and Related Work

The Road goes ever on and on
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can,...

—J.R.R. Tolkien
The Lord of the Rings

The work in this dissertation makes significant advances in automated formal MCM verification of hardware. The background information in this chapter provides an introductory overview of MCM verification research. Each section also cites the relevant related work for the topic under discussion.

Section 2.1 covers MCMs for hardware ISAs, including the relationship between microarchitectural features and the choice of ISA-level MCM. Section 2.2 provides an overview of the formal verification approaches relevant to this dissertation, with a focus on model checking. Section 2.3 covers prior work and approaches for formal MCM specification and verification, apart from the PipeCheck work [LPM14,LSMB16]. PipeCheck is a starting point for much of the work in this dissertation, and thus is covered in detail in Section 2.4. The chapter concludes with a summary (Section 2.5).

Core 0	**Core 1**
(i1) [x] ← 1	(i3) [y] ← 1
(i2) r1 ← [y]	(i4) r2 ← [x]
SC forbids r1=0, r2=0	

Figure 2.1: Code for litmus test `sb`

Core 0	**Core 1**
(i1) [x] ← 1	(i3) r1 ← [y]
(i2) [y] ← 1	(i4) r2 ← [x]
SC forbids r1=1, r2=0	

Figure 2.2: Code for litmus test `mp`

2.1 Memory Consistency Model (MCM) Background

2.1.1 Litmus Tests

To illustrate or verify MCM properties, computer architects, programmers, and researchers often use small 4-8 instruction programs called *litmus tests*, and this dissertation follows suit. MCM scenarios of interest can almost always be expressed by such small tests. In a litmus test, the initial values of all memory addresses are assumed to be 0 by convention. Litmus tests are usually designed so that one of their outcomes (i.e. one set of values for their loads) is forbidden by the MCM under consideration. A given litmus test also often corresponds to some hardware or compiler feature that affects MCM behaviour. In this dissertation (and often in the literature), a reference to a litmus test refers to the forbidden outcome of that litmus test unless otherwise specified.

The `sb` (store buffering) program in Figure 2.1 is an example of a litmus test. It is essentially the synchronization algorithm from Section 1.2 reduced to its minimal form. To briefly reiterate the explanation of this program from Section 1.2, cores 0 and 1 each write 1 to addresses `x` and `y` respectively, and then read the value of the address written by the other core. If no reordering or buffering of memory operations occurs, then it is impossible for both loads (`i3` and `i4`) to return 0. Thus, the outcome `r1=0,r2=0` is forbidden under SC.

However, on most processors today (including all commercial ones), the outcome `r1=0,r2=0` is allowed and in fact observable if the test were to be run on the processor. The reason for this is that most processors today have per-core store buffers. As discussed in Section 1.2, the use of per-core store buffers results in stores being reordered with subsequent loads from the point of view of other cores, allowing both loads `i3` and `i4` to return 0.

Another litmus test used repeatedly in this dissertation is the "message passing" or `mp` litmus test (Figure 2.2). In `mp`, core 0 writes to a data address `x` and then to a flag `y` to indicate that the data is ready. Core 1 reads the value of the flag `y` (to check whether the data in `x` is valid) and then reads the value of the data `x`. Under SC, it is forbidden for the load of `y` to return 1 while the load of `x` returns 0. In other words, SC forbids core 1 to see the update of the flag `y` without also observing the update to the data `x`.

Litmus tests have been used for MCM analysis for decades, even in Lamport's original sequential consistency paper [Lam79]. However, the term "litmus test" for such minimal program examples is somewhat more recent [OSS09]. Litmus tests are widely used in the literature to illustrate specific MCM scenarios, as `sb` does above for the microarchitectural feature of store buffering. They are widely used throughout this dissertation for the same function. Litmus tests can be used for bug-finding in MCM implementations through dynamic analysis (Section 2.3.5). They are also often used to dynamically validate formal ISA-level MCM specifications during their creation, if the processor exists prior to the creation of the formal MCM specification (Section 2.3.1). Litmus tests can also be formally analysed to determine whether they are allowed or forbidden under a given ISA-level MCM [AMT14, SSA+11], a programming language MCM [BOS+11a, BDW16], or a microarchitectural ordering specification [LPM14, LSMB16].

Litmus tests can be generated in a number of ways, including by hand (like `sb` above) or through random generation [diy12]. Schemes for generating better suites of litmus tests have also been developed [AMSS10, MHAM11, LWPG17]. One automated litmus test synthesis tool [LWPG17] can, when given a formal MCM specification, generate all litmus tests (up to a bounded number of instructions) for that MCM that satisfy a *minimality criterion*. The minimality criterion requires that no synchronization mechanism in the test can be weakened without causing new behaviours to become observable. As a result, the generated tests provide excellent coverage of MCM corner cases.

A number of ISA-level MCMs can be differentiated from each other by using a set number of litmus tests [MHAM11]. However, it is still an open question as to whether the verification of a microarchitecture's MCM correctness across all programs can be reduced to the verification of its MCM correctness for a set number of litmus tests. Chapter 5 shows how to sidestep this problem and prove microarchitectural MCM correctness across all programs without using litmus tests at all.

2.1.2 Speculative Implementations of MCMs

As Section 1.2 covers, today's microarchitectures are at odds with an intuitive MCM like SC. Microarchitectural features like out-of-order execution and coalescing store buffers can easily cause SC violations. (Appendix A details various microarchitectural features that can affect MCM behaviour, including litmus test examples.) One way to keep using microarchitectural features that break SC is to implement them *speculatively*. In other words, the hardware attempts to reorder memory accesses but makes sure to check if any other core can observe the reordering. This can happen if another core is accessing the same address as one of the speculatively reordered accesses during the speculation, and at least one of the two accesses to that address is a write. Such an occurrence is known as a *conflict*. For example, in the `mp` litmus test (Figure 2.2), if core 0 speculatively reordered the stores `i1` and `i2` to addresses `x` and `y`, then this reordering could potentially conflict with the loads `i3` and `i4` to `y` and `x` on core 1. If the speculating core conflicts with another core, one or both[1] cores roll back the execution of their speculatively reordered accesses and attempt to execute them again. The processor's coherence protocol (Section A.5) is frequently used to detect conflicts, as it keeps track of which cores have access to which data. As long as there are relatively few conflicts, the hardware will be able to reorder the memory accesses most of the time and gain the associated performance benefits, while still appearing to implement SC for any assembly language program.

There has been much work on speculatively reordering memory operations while implementing a strong MCM. Hill [Hil98] argues that multiprocessors should support SC because the additional performance benefits of weak MCMs over the speculative execution of SC do not outweigh the counterintuitive outcomes and lower programmability of weak MCMs. Gharachorloo et al. [GGH91] proposed two speculative techniques for improving performance under any MCM. The techniques were speculative execution of future loads and non-binding prefetches[2] for future accesses before their execution became legal. Ranganathan et al. [RPA97] allowed speculative commits of loads and later instructions before outstanding stores, and used a history buffer to recover from consistency violations. Gniady et al. [GFV99] allowed both loads and stores to bypass each other speculatively, and also used a history buffer to aid in recovery from consistency violations. BulkSC [CTMT07] speculatively implements SC by dynamically grouping sets of consecutive instructions into chunks that appear to execute atomically and in isolation. This enables the reordering and overlapping of memory access within chunks and across chunks. A chunk which conflicts with another

[1]Each core may be speculatively reordering its own memory operations at the same time.
[2]Data fetched by a non-binding prefetch may be invalidated before it can be used.

chunk leads to one of them being rolled back and re-executed. InvisiFence [BMW09] can speculatively implement MCMs with low storage requirements and without the global arbitration for chunk commits that BulkSC requires.

Transactional Memory (TM) [HM93] enables assembly language programs to define transactions (analogous to database transactions) that either commit as a single atomic unit or abort and do not write any data. Hardware keeps track of conflicts between transactions, allowing them to commit or requiring that they abort as necessary. Transactions on different cores may overlap with each other, and are only ordered with respect to each other if they conflict. This improves memory throughput, while also making lock-free synchronization[3] mechanisms easy and efficient to use. Later speculative approaches like BulkSC are similar to TM in that they commit memory operations in sets rather than one at a time. However, a key difference between the two is that in TM the programmer defines the size of the transactions, while in approaches like BulkSC the size of the chunks is determined by the hardware. Transactional memory saw much subsequent research over the years, including implementations in software and hardware. Harris et al. [HLR10] provides an overview of research in this area.

While speculative implementations of MCMs can achieve the performance benefits of weak MCMs while presenting a strong MCM to the programmer, they also incur substantial additional hardware complexity to keep track of the speculation. This can make such designs more prone to bugs, as was the case for Intel's recent implementation of transactional memory in its processors [Hac14, Int20]. Furthermore, speculatively executing memory operations before it is valid to do so under the processor's MCM can also lead to side-channel attacks [TLM18b]. Rather than adopting a purely speculative approach, manufacturers of commercial processors have instead chosen instead to

[3]Lock-free data structures can be accessed without locks or mutexes, i.e., they are thread-safe.

MCM	**Relax W→R**	**Relax W→W**	**Relax R→R**	**Relax R→W**	**WA**	**Safety Nets**
TSO [SPA92, OSS09]	✓				rMCA	RMWs and `MFENCE` on x86
PC [Goo89]	✓				nMCA	RMWs
PSO [SPA92]	✓	✓			rMCA	RMWs and `STBAR` fence
RMO [SPA94]	✓	✓	✓	✓	rMCA	`MEMBAR` fences
Power [SSA+11, AMT14]	✓	✓	✓	✓	nMCA	Fences including `sync` and `lwsync`
ARMv7 [AMT14]	✓	✓	✓	✓	nMCA	`dmb` and other fences
ARMv8 [PFD+18]	✓	✓	✓	✓	rMCA	`dmb` and other fences, plus `ldar` and `stlr`
RISC-V [RIS19]	✓	✓	✓	✓	rMCA	`FENCE` instruction and optional `AMO`s
WO [AH90]	✓	✓	✓	✓	rMCA	synchronization instructions and RMWs
RCsc [GLL+90]	✓	✓	✓	✓	rMCA	release and acquire instructions and RMWs
RCpc [GLL+90]	✓	✓	✓	✓	nMCA	release and acquire instructions and RMWs

Table 2.1: Summary of Weak MCMs. The write atomicity (WA) abbreviations are explained in Section 2.1.3. RMWs are read-modify-write instructions.

Core 0	**Core 1**
(i1) [x] ← 1	(i4) [y] ← 1
(i2) `MFENCE`	(i5) `MFENCE`
(i3) r1 ← [y]	(i6) r2 ← [x]
TSO forbids r1=0, r2=0	

Figure 2.3: Code for x86-TSO litmus test `sb+fences`

implement *weak* or *relaxed* MCMs that allow some behaviours forbidden by SC. The next section discusses such MCMs.

Core 0	**Core 1**	**Core 2**
(i1) [x] ← 1	(i2) r1 ← [x]	(i5) r2 ← [y]
	(i3) <fence>	(i6) <fence >
	(i4) [y] ← 1	(i7) r3 ← [x]
Cumulative fence `i3` forbids r1=1, r2=1, r3=0		

Figure 2.4: Code for litmus test `wrc+fences`

2.1.3 Weak/Relaxed MCMs

Most processors today (including all commercial ones) have chosen to implement *weak* or *relaxed* MCMs that do not require one or more of the orderings required by SC. This enables architects to utilise microarchitectural features such as those in Appendix A and have them be programmer-visible. Programmer-visible relaxations fall into two main categories: relaxations of *program order* and relaxations of *write atomicity*.

Program order relaxations relax the ordering between certain pairs of memory instructions in a program. For instance, a weak MCM may relax Store→Store ordering, allowing store instructions to be reordered with subsequent stores in the program. This would allow a processor to perform `i1` and `i2` out of order in `mp` (Figure 2.2), resulting in the outcome `r1=1,r2=0` (which is forbidden under SC) becoming observable.

Meanwhile, relaxations of write atomicity dictate whether a store must be observed by every core in the system at the same (logical) time. A store is *multi-copy-atomic* [Col92] (referred to as MCA in this dissertation) if it becomes visible to all cores in the system at the same time. Some weak MCMs may allow a core to "read its own write early", in which case a store can become visible to the writing core before it becomes visible to all other cores. In this case, the store must become visible to all cores other than the writing core at the same time. This variant of write atomicity is referred to as rMCA in this dissertation. Finally, other weak MCMs may allow a store to become visible to cores other than the writing core at different times. This is known as "non-multi-copy-atomicity", and is referred to as nMCA in this dissertation. (Section A.4 explains write atomicity in detail, including some of the hardware optimizations enabled by allowing such relaxations.)

Architectures that have weak MCMs provide one or more "safety net" instructions [AG96] in their ISA that can be used to enforce ordering between relaxed memory operations where necessary (for instance, in synchronization code). These are generally one of two types: (i) *fences* and (ii) *release* and *acquire* instructions. Fences order instructions preceding the fence before those after it. Release and acquire instructions are special store and load operations (respectively) that enforce ordering with respect to other memory operations. They are conceptually analogous to lock releases and acquires in parallel programs.

Table 2.1 summarises the major characteristics of various well-known weak MCMs for general-purpose processors. This table draws from the table of MCMs in Adve and Gharachorloo [AG96] while incorporating changes since that work was published. I now provide a brief overview of these MCMs.

Weak MCMs Used in Commercial Processors

TSO (Total Store Order) [OSS09] is the MCM of Intel and AMD x86 processors as well as certain SPARC processors. TSO relaxes Store$\rightarrow$Load ordering and implements rMCA write atomicity, but preserves all other orderings. Under x86, programmers can add an `MFENCE` between a store and a load to enforce ordering between them where necessary. For example, one can forbid the execution of `sb` where `r1=0,r2=0` by adding `MFENCE`s between each pair of instructions, as in Figure 2.3. Programmers can also use read-modify-write instructions (RMWs) to enforce ordering between stores and loads [AG96]. PC (Processor Consistency) [Goo89] also relaxes Store$\rightarrow$Load ordering but implements the weaker nMCA write atomicity. It preserves all other orderings, and programmers can enforce ordering through RMWs where necessary.

PSO (Partial Store Ordering) [SPA92] was developed by SPARC, and relaxes both Store→Load and Store→Store ordering. It implements rMCA write atomicity. Programmers can use the `STBAR` fence and RMWs to enforce ordering where necessary. RMO (Relaxed Memory Order) [SPA94] was also developed by SPARC, and relaxes program order between all Load and Store pairs. It provides `MEMBAR` fences to enforce ordering where needed.

IBM Power processors implement one of the weakest MCMs. The Power MCM relaxes all program orders for accesses to different addresses, and also implements weaker nMCA write atomicity. Programmers can enforce ordering where necessary using a number of fences, including the `sync` and `lwsync` fences. The MCM of ARMv7 processors is very close to that of Power, also relaxing all program orders and implementing nMCA. It also uses fences to enforce ordering, the most common of which is the `dmb` fence. Alglave et al. [AMT14] discuss differences between the Power and ARMv7 MCMs.

For ARMv8, the subsequent iteration of the ARM ISA, ARM strengthened the MCM of its processors, making them implement rMCA. They also added the release and acquire instructions `stlr` and `ldar` (explained below) to give programmers the option of synchronizing using these instructions. The RISC-V MCM is very close to that of ARMv8; it also relaxes all program orders and implements rMCA. RISC-V includes a `FENCE` instruction to enforce ordering between memory operations where necessary. RISC-V also has an optional "A" (Atomic) ISA extension that includes release-acquire RMW operations (called `AMO`s). (Note that "the RISC-V MCM" here refers to the ratified RISC-V MCM which fixes the issues identified by TriCheck [TML+17].)

Weak Ordering and Release Consistency

Release and acquire instructions were first introduced in the RC (Release Consistency) [GLL+90] and Weak Ordering (WO) [DSB86, AH90] MCMs. On architectures

Core 0	**Core 1**
(i1) [x] $\leftarrow$ 1	(i4) r1 $\xleftarrow{acq}$ [y]
(i2) [y] $\xleftarrow{rel}$ 1	(i5) r2 $\leftarrow$ [x]
(i3) [x] $\leftarrow$ 2	
RC forbids r1=1, r2=0	

Figure 2.5: Code for an RC (release consistency) variant of `mp` where `i2` is a release (denoted by $\xleftarrow{rel}$) and `i4` is an acquire (denoted by $\xleftarrow{acq}$).

that support them, release and acquire operations provide an alternative to fences for inter-core synchronization under weak MCMs. Releases and acquires allow programmers to specify which memory accesses are used for synchronization. The remaining accesses are considered to be data accesses. Consistency only needs to be enforced at synchronization points between cores, so the hardware can leverage the labelling of synchronization reads and writes to complete data reads and writes faster [AH90]. RC and WO operate on the same principles; this section utilises RC's terminology to explain their workings.

Acquire and release operations are conceptually analogous to acquire and release operations on a lock used to enforce mutual exclusion for a shared resource in a parallel program. In parallel programs that share data, a thread generally acquires a lock, operates on the shared data protected by the lock in its critical section, and then releases the lock. The critical section is guaranteed to occur between the lock and unlock. In a similar vein, acquire operations are loads, and must be ordered before all accesses after the acquire in program order. Meanwhile, release operations are stores, and must be ordered after all accesses before the release in program order. If an acquire observes the write of a release operation, it must observe all accesses preceding that release, enabling synchronization between threads. Read-modify-write instructions can be labelled as both acquires and releases [GLL$^+$90]. Under the RCsc variant of RC, releases and acquires are required to be sequentially consistent with each other, while under RCpc, releases and acquires must obey PC (processor consistency) with respect to each other. RCpc is thus weaker than RCsc.

For an example of using acquires and releases under RC, consider Figure 2.5's variant of the `mp` litmus test. Under RC[4], if all instructions in the test were data accesses, the outcome `r1=1,r2=0` would be allowed as RC relaxes all program orders and no synchronization instructions have been used. However, since `i2` is labelled as a release and `i4` is labelled as an acquire, the outcome `r1=1,r2=0` becomes forbidden. The release `i2` ensures that `i1` is ordered before `i2`, while the acquire `i4` ensures that `i4` is ordered before `i5`. Thus, when the acquire `i4` observes the release `i2`, the data accesses before `i2` (like `i1`) are ordered before the data accesses after `i4` (like `i5`). Thus, if `i4` returns 1, then `i5` is guaranteed to see the store `i1` to `x`, forbidding the outcome `r1=1,r2=0`.

Release and acquire operations are different from fences in that they are one-way barriers. Fences ensure that memory operations preceding the fence are ordered before memory operations after the fence. However, acquires and releases only enforce order on one set of memory operations with respect to themselves. Acquires order succeeding accesses, while releases order preceding accesses. Thus, accesses after a release are free to be reordered with the release, and accesses before an acquire are free to be ordered with respect to the acquire. For example, in Figure 2.5, `i3` is free to be reordered with the release `i2`. This is known as *roach-motel movement*[5], and intuitively corresponds to making a critical section larger [BA08]. Roach-motel movement is beneficial as it allows some reordering of memory operations across barriers while not affecting the correctness guarantees of acquire and release operations.

The SC-for-DRF Guarantee

Application and systems software programmers would much prefer to reason about their programs under the intuitive model of SC, as programming under weak MCMs is

[4]The reasoning for this litmus test applies to both RCsc and RCpc.

[5]The term arises from a cockroach trapping product named "Roach Motel" [Man07]. Cockroaches can enter the trap but they cannot exit it. Similarly, under roach-motel movement, memory accesses can enter a critical section but cannot exit it.

very difficult. However, programmers cannot restrict themselves to SC hardware as all commercial processors implement weak MCMs. The SC-for-DRF guarantee[6] [AH90, GLL$^+$90] solves this problem. It states that if programmers include a certain amount of synchronization in their program—specifically, enough synchronization to eliminate all data races[7] in the program's SC executions—then it will behave as if it were running on a sequentially consistent system, even if the underlying processor implements a weak MCM (assuming it does so correctly). Thus, programmers can reason about their program under SC, while also reaping the performance benefits of weak MCMs. The SC-for-DRF guarantee also forms the basis for the MCMs of high-level programming languages.

Despite its importance, the SC-for-DRF guarantee is not a silver bullet for MCM issues in parallel systems. A parallel system that purports to obey the SC-for-DRF guarantee will only restrict itself to SC behaviours for DRF programs if the hardware obeys its MCM. Buggy hardware will still cause incorrect executions regardless of whether the program has data races or not, necessitating MCM verification research such as the work in this dissertation.

Summary

This section and Table 2.1 provide a high-level overview of weak MCMs today. The details of such MCMs can be much more nuanced, as Section 2.1.4 shows. To enable more comprehensive MCM specification and verification, the past decade has seen much work on developing and formalising the specifications of these weak ISA-level MCMs (Section 2.3.1).

[6]This guarantee was also shown as part of the *properly-labelled* approach of Gharachorloo et al. [GLL$^+$90].

[7]Intuitively speaking, a race in an execution consists of two accesses (at least one of which is a write) running on different cores/threads which access the same address and which have no synchronization accesses occurring between them [SHW11]. A data race is a race where at least one of the accesses is to data, rather than for synchronization.

Core 0	Core 1
(i1) [x] ← 1 (i2) dmb (i3) [y] ← 1	(i4) r1 ← [y] <ctrlisb> (i5) r2 ← [x]
ARMv7 forbids: r1=1, r2=0	

Figure 2.6: Code for ARMv7 litmus test `mp+dmb+ctrlisb`. In the above code, "<ctrlisb>" represents an instruction sequence comprising a control dependency (Section A.3) followed by an `isb` fence. Such an instruction sequence is sufficient to enforce local ordering between two load instructions on ARMv7.

Core 0	Core 1
(i1) [x] ← 1 (i2) dmb (i3) [y] ← 1	(i4) r1 ← [y] (i5) [y] ← 2 (i6) r2 ← [y] <ctrlisb> (i7) r3 ← [x]
ARMv7 observable: r1=1, r2=2, r3=0	

Figure 2.7: Code for ARMv7 litmus test `mp+dmb+fri-rfi-ctrlisb`, which demonstrates a highly counterintuitive MCM outcome. This behaviour was observed on an ARMv7 processor and considered desirable by architects [AMT14], despite the similar `mp+dmb+ctrlisb` test (Figure 2.6) being forbidden. Once again, "<ctrlisb>" represents an instruction sequence comprising a control dependency (Section A.3) followed by an `isb` fence.

2.1.4 The Need for Formal MCM Specification and Verification

Hardware features can combine to create counterintuitive MCM outcomes in numerous ways. Architects may desire to allow such outcomes for performance reasons or forbid them to enhance programmability. Therefore, unambiguous specifications for ISA-level MCMs are necessary to clearly define which outcomes are allowed and which are forbidden. Likewise, stringent verification methods are also necessary to ensure that hardware respects its MCM specification.

Consider Figure 2.6's ARMv7 litmus test `mp+dmb+ctrlisb`, which is forbidden on ARMv7. The `dmb` fence enforces that all cores observe the store to `x` (`i1`) before the store to `y` (`i3`), while the `ctrlisb`[8] on core 1 enforces that the loads `i4` and `i5` execute in order. The combination of these orderings make it impossible for core 1 to observe the write to `y` before the write to `x` (i.e. for the outcome `r1=1,r2=0`) to occur.

Now consider Figure 2.7's variant of this test, called `mp+dmb+fri-rfi-ctrlisb`. Here, `i1` and `i3` are still ordered by a `dmb`. The loads to `y` (`i4`) and `x` (`i7`) are still separated by a `ctrlisb`, with an additional write to and read from `y` in between `i4` and the `ctrlisb`. Since ARMv7 preserves program order for accesses to the same address, one would expect that `i4` would still be ordered before `i7`, which would make the outcome `r1=1, r2=1, r3=0` forbidden. However, this outcome was in fact observable on ARMv7 hardware [AMT14], and the corresponding architects even considered it desirable behaviour. An intuitive explanation of why the outcome is observable is as follows. At the start of execution, core 1 can observe that `i5` and `i6` are a store and load to the same address that are next to each other in program order. Thus, it should be allowed for the load `i6` to read from the store `i5` (i.e., for `r2=2`). Critically, this can be done before `i4` is executed, as `i4`'s result no longer affects the result of `i6` (since `i6` reads from `i5`, which is after `i4` in program order). Once `i6` has executed, `i7` is free to execute as doing so will not break the dependency between `i6` and `i7`. Thus, `i7` can now read the initial value of 0 for `x` (`r3=0`). Finally, `i1` and `i2` can be stored in order, and observed by core 1, allowing `i4` to return `r1=1`.

The above example is just one of the numerous ways in which hardware features and optimizations can combine to generate counterintuitive outcomes due to MCM issues. Simply stating whether a few litmus tests are allowed or forbidden is not sufficient to specify the MCM behaviour of a parallel system. As seen above, even relatively small changes to a forbidden litmus test can unexpectedly cause it to become observable. A *formal specification* (i.e. one written in some form of mathematical logic)

[8]`ctrlisb` represents an instruction sequence comprising a control dependency (Section A.3) followed by an `isb` fence. Such an instruction sequence is sufficient to enforce local ordering between two load operations on ARMv7.

of the MCM of a parallel system is necessary to comprehensively and unambiguously define which MCM behaviours are allowed and which are forbidden. Once the system's MCM is formally specified, any litmus test can be formally evaluated against it to conclusively determine whether it is allowed or forbidden.

Similarly, formal verification is critical to ensure that hardware and software respect their MCMs. As Section 1.3 covered, today's multiprocessors are nondeterministic, meaning that even if one run of a litmus test on a parallel system satisfies the MCM requirements of the system, another run of the same test may violate the system's MCM. Formal verification of the parallel system for the litmus test can cover all possible executions of the litmus test, and is necessary to ensure that the system will always run the litmus test correctly. Furthermore, a system must respect its MCM for all programs, of which there are an infinite number. It is impossible to explicitly test each possible program on a system, so formal verification is necessary to ensure that a system correctly implements its MCM across all programs.

A number of MCM bugs have been found in parallel systems in recent years [Hac14, Int20, AMT14, ARM11, ABD+15, SD16, TML+17], further highlighting the necessity of formal MCM verification. The work in this dissertation also discovers a real-world MCM bug in open-source Verilog (Chapter 3). This adds further weight to the argument for formal MCM verification and showcases the efficacy of the work in this dissertation.

The next section covers the basics of formal verification, with a focus on model checking (the verification approach on which the work in this dissertation is based). Sections 2.3 and 2.4 then cover prior work on formal MCM specification and verification, including necessary background on such work.

2.2 Formal Verification Background

The overarching idea behind formal verification is to represent a hardware and/or software system using a mathematical model, and then prove the correctness of the model for a mathematically specified property. The proof may use mathematical techniques like induction, or it may simply comprehensively check all possible cases of the model for the property under verification. Formal verification can provide strong correctness guarantees because it is based on mathematical proofs. For instance, formal verification of a multicore hardware design can prove that a litmus test will never be observable on the design, even though the hardware is nondeterministic.

Section 2.2.1 covers *model checking*, which is the automated formal verification method underlying the work in this dissertation. Other related work on MCM verification involves the manual writing of proofs in proof assistants, which are discussed in Section 2.2.2. Finally, Section 2.2.3 covers the two major modelling styles for formal MCM specifications: *operational* and *axiomatic*, each of which has their own benefits and drawbacks.

2.2.1 Model Checking

Model checking is a formal verification technique that explores all possible states of a system to check whether it satisfies a certain property [CE81, BK08]. If the system does not satisfy the property, model checking returns a *counterexample*: an execution of the system that does not satisfy the property. All the work presented in this dissertation utilises model-checking based approaches to conduct verification.

Model checking requires a logic-based representation of the system being verified as well as the property to verify on that system. This is traditionally done using *transition systems* and *temporal logic* respectively. Model checking may explicitly search through the states of a system to see if a property holds (*explicit-state model checking*). Model

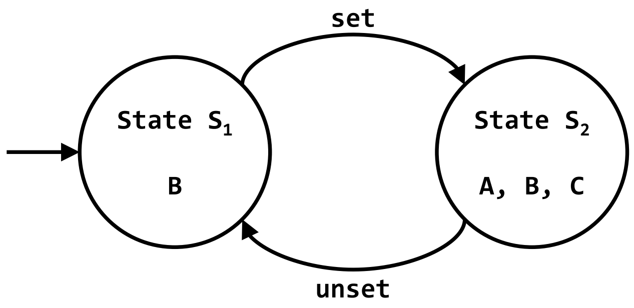


Figure 2.8: Pedagogical transition system with two states S_1 and S_2. Each state is annotated with the atomic propositions that are true in that state.

checking can also be conducted using techniques based on Binary Decision Diagrams (BDDs) [BCM+92] and Boolean satisfiability (SAT) solvers [BCCZ99] (*symbolic model checking*), which can have significant performance benefits over explicit-state model checking. Finally, Satisfiability Modulo Theories (SMT) solvers [BSST09] build on SAT-based model checking to enable the specification of systems and properties using more expressive *theories*. This has benefits in terms of both expressiveness and performance over standard SAT-based model checking. All the work in this dissertation utilises SAT or SMT-based model checking approaches to conduct verification.

Specifying Systems and Properties [BK08]

In model checking, the system being verified is traditionally represented as a state machine or transition system. A transition system is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

- S is a set of states
- Act is a set of actions
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation. It links each state S_i with the next state that the system will transition to if a given action is taken while in S_i.
- $I \subseteq S$ is a set of initial states

- AP is a set of atomic propositions (i.e. Boolean variables)
- $L : S \rightarrow 2^{AP}$ is a labelling function that when passed a given state, returns the set of atomic propositions that are true in that state

Figure 2.8 shows an example pedagogical transition system. This system has two states. The left state S_1 is the initial state (indicated by a $\rightarrow$ leading into it with no source), where atomic propositions A and C are false, and B is true. This state can transition to the state S_2 on the right using the *set* action. In S_2, A, B, and C are all true. S_2 can transition to S_1 using the *unset* action. A state in a transition system is terminal if it has no state that it can transition to. (A state is not terminal even if it can only transition to itself.) Parallel reactive[9] systems are best modelled by transition systems without terminal states [BK08], and the discussion of transition systems in this dissertation is restricted to such transition systems.

In a transition system without terminal states, a *path* is an infinite state sequence $s_0s_1s_2...$ such that $s_0 \in I$ (i.e. s_0 is an initial state) and $s_i \in Post(s_{i-1})$[10] for all $i > 0$. A *trace* of a path $\pi = s_0s_1s_2...$ is $L(s_0)L(s_1)L(s_2)...$ and is denoted by $trace(\pi)$. In other words, the trace is the sequence of sets of atomic propositions that are valid in the states of the path.

The property to verify on the system is traditionally specified in temporal logic, a logic which extends propositional logic with capabilities for referring to the behaviour of a system over time. For instance, temporal logic allows properties like "P is always true on every trace" and "Q will eventually be true on every trace" to be easily stated. There are a number of different temporal logics, including LTL (Linear Temporal Logic) [Pnu77], CTL (Computation Tree Logic) [CE81], and CTL* [EL87]. LTL forms

[9]A reactive system is one which is repeatedly prompted by the outside world and whose role is to continuously respond to external inputs [HP89]. A processor can be thought of as a reactive system; the instructions it is given to execute constitute some of its external inputs.

[10]$Post(s)$ is the set of states that s can transition to.

the basis for the SystemVerilog Assertions used by the work in Chapter 3, and is explained in detail in Section 3.4.1.

The system and property for a given model checking problem may also be represented in other ways, such as through a set of axioms (invariants) that the system respects. `herd` (Section 2.3.1) and PipeCheck (Section 2.4) are examples of such axiomatic modelling frameworks. See Section 2.2.3 for a comparison of axiomatic models to operational models (which use transition systems) for formal MCM analysis. Axiomatic models are used for the majority of the work in this dissertation.

Explicit State Model Checking [BK08]

Model checking may be conducted by explicitly searching through the states of a transition system to determine whether it implements a temporal logic property. As a simple pedagogical example, consider verifying the property "A is always false" on the transition system in Figure 2.8. A verification procedure could check such a property by conducting a depth-first search of the transition system's state graph from each initial state, and searching for a state where A is true. (In a transition system's state graph, the successors of a given state are the possible states that it can transition to.) The search would start at the initial state S_1, where A is false as required. Then the search would check the successors of S_1, namely state S_2. A is true in S_2, violating the property, and so the procedure would return a trace starting with $s_0 s_1$ as a counterexample. On the other hand, if verifying the property "B is always true" on the same transition system, a search would first check S_1 and then S_2, and find that B is true in both states. Since there are no more states to examine, the search would return that the property holds.

Generally speaking, explicit-state model checking of temporal logic properties is somewhat more complicated, requiring acceptance checks for Büchi automata, a particular type of finite-state machine [BK08]. Section 3.4.1 provides a detailed explanation of explicit-state model checking for general LTL properties, as the algorithm for doing so is relevant to the RTLCheck work in Chapter 3.

Symbolic Model Checking [BK18]

In contrast to explicit-state methods, model checking may also be conducted *symbolically* through BDDs and SAT solvers. The JasperGold verifier used by the RTLCheck work in Chapter 3 utilises BDD and SAT-based model checking approaches [Cad16]. Furthermore, SAT-based model checking forms the basis for SMT-based model checking (discussed after this section), which is the verification method used by the majority of the work in this dissertation. This section focuses on SAT-based model checking, as the details of BDD-based model checking are not relevant to the work presented in this dissertation.

Symbolic model checking represents the state sets and transition relation of a transition system using expressions in a symbolic logic like propositional logic. This can dramatically reduce the size of the data structures for representing state sets, and can potentially result in order-of-magnitude improvements in verification performance [CHV18].

SAT solvers can be used to conduct *bounded model checking* (BMC) [BCCZ99]. In BMC, a symbolic representation of a transition system and a property are jointly unwound (i.e. instantiated) for k steps, giving a formula that is satisfiable if and only if there is a counterexample to the property that has length k. This formula is then passed to a SAT solver. If the solver returns that the formula is unsatisfiable, no counterexamples of length k exist. On the other hand, if the solver finds a satisfying assignment, that assignment corresponds to a counterexample of length k.

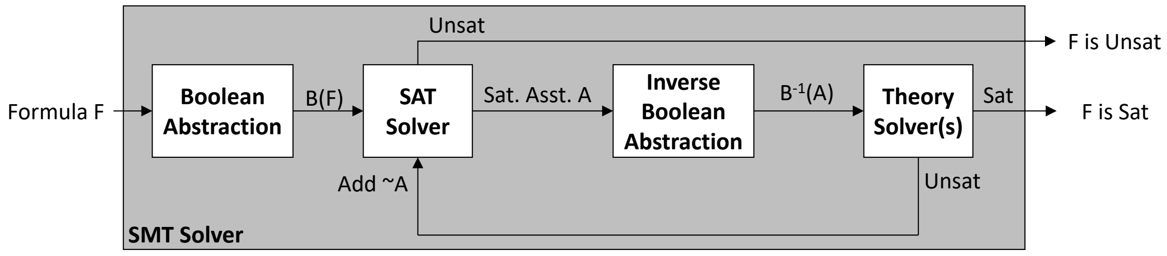


Figure 2.9: Block diagram of an SMT (Satisfiability Modulo Theories) solver, adapted from Gupta [Gup17] with some visual presentation changes.

For example, if searching for counterexamples of length k to a property "p is always true on every trace" using BMC, the verification of the property can be encoded as follows[11]:

$$\exists s_0, ..., s_k . I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \left(\bigvee_{i=0}^{k} \neg p(s_i) \right) \tag{2.1}$$

Bounded verification such as this can provide guarantees that a system satisfies a property for all executions of length up to and including some bound k. By itself, BMC cannot provide *unbounded verification*, i.e., a guarantee that there exist no counterexamples to the property of any length. However, BMC can use inductive techniques, Craig interpolation, and other techniques to achieve unbounded (i.e. complete) verification [BK18].

The methodologies and tools presented in this dissertation all conduct bounded verification except for PipeProof (Chapter 5), which conducts unbounded verification.

Model Checking Using SMT [BSST09, BT18]

Model checking using SMT (Satisfiability Modulo Theories) solvers is the verification method used by Chapters 4 and 5 of this dissertation. It builds on SAT-based model checking by adding the ability to express properties and systems using various theories, which provide improvements in expressibility and verification performance.

[11]$p(s_i)$ denotes whether p holds in state s_i.

Propositional logic is capable of efficiently specifying a variety of properties. However, certain types of properties are more naturally and compactly specified in other forms of logic. Consider for instance the following pedagogical set of constraints:

$$(x = 5) \wedge ((y = x + 3) \vee (y = x - 2)) \wedge (y = 2x) \tag{2.2}$$

This formula does not naturally map to propositional logic because its individual atoms are not Boolean variables, but linear equations with algebraic variables and values. However, it can be modelled through the combination of propositional logic and an additional logical *theory*. For instance, the Linear Integer Arithmetic (LIA) theory can reason about linear arithmetic over integers, and it can be used to help create a model for the above formula. Such models can be verified using SMT: the combination of a SAT solver with solvers for one or more logical theories.

Figure 2.9 shows a block diagram of the basic version of an SMT solver. To solve an SMT formula, the individual atoms corresponding to theory formulae are abstracted by Boolean variables, resulting in a normal SAT formula. This SAT formula is then solved by the SAT solver inside the SMT solver. If the SAT formula is unsatisfiable, then the SMT formula it abstractly represents is unsatisfiable too. On the other hand, the SAT solver may find a satisfying assignment for the SAT formula. In this case, the SMT solver needs to check that the assignment is also valid under the theories being used. The abstract Boolean variables in the assignment are translated back into the relevant theory formulae, and the theory solvers check that they represent a valid assignment in the relevant theories. If they do, then the overall SMT formula is satisfiable. Otherwise, the theory solvers notify the SAT solver that the satisfying assignment it found is invalid, and the SAT solver attempts to find another satisfying assignment to the abstracted formula.

For example, consider trying to check the satisfiability of Formula 2.2. We can abstract $x = 5$ by M, $y = x + 3$ by N, $y = x - 2$ by P, and $y = 2x$ by Q, giving us $M \wedge (N \vee P) \wedge Q$. The SAT solver will return a satisfying assignment such as $M \wedge N \wedge Q$. M, N, and Q are then translated back into their corresponding equations, and the LIA theory solver is asked to check whether $(x = 5) \wedge (y = x + 3) \wedge (y = 2x)$ can be true. It is clearly impossible to satisfy these constraints (y cannot be both 10 and 8), so the LIA solver informs the SAT solver that its satisfying assignment is invalid, and asks it to find another. The SAT solver then returns $M \wedge P \wedge Q$, which is equal to $(x = 5) \wedge (y = x - 2) \wedge (y = 2x)$, which is also impossible for the theory solver to satisfy. The SAT solver is then asked to find yet another satisfying assignment, but no other satisfying assignment exists. Thus, the SMT solver returns that the overall formula is unsatisfiable.

2.2.2 Interactive Proof Assistants

A proof assistant is a tool that aids users in writing proofs of theorems that they can specify in the proof assistant. It keeps track of what techniques have been used in the proof and what remains to be proven. Most importantly, though, the proof assistant is capable of verifying whether the user's proof does indeed prove the theorem it purports to, using an approach based on programming language type checking [Chl13]. This allows users of proof assistants to ensure that their proof of a system's correctness does in fact prove that the system is correct.

Prior work has developed approaches that enable hardware and software to be written in such proof assistants. Users can then prove the correctness of this hardware and software using the proof assistant, and then automatically extract a verified implementation from the proof assistant. Kami [CVS$^+$17] is an example of such a flow. (Kami is discussed further in Section 2.3.4.) However, writing proofs in proof assistants requires formal methods expertise and can require significant manual effort, creating a high barrier to entry. As a result, the usage of proof assistants is not as amenable to use by typical hardware and software engineers as approaches like model checking. A number of proof assistants exist, including Coq [Coq04], Isabelle [Isa20], and Lean [Lea20].

2.2.3 Operational and Axiomatic Models

There are two major modelling styles for formal hardware and software MCM specifications, namely operational and axiomatic models. Models like the transition system in the example of Section 2.2.1 are known as operational models. They are defined as state machines, and when used to model hardware, they often resemble the machine that they model. An operational model explicitly describes its allowed behaviours through its states and the possible transitions among them. In recent years, prior work has used operational models to model hardware for MCM analysis [VCAD15, CVS+17, SSA+11, GKM+15, FGP+16, FSP+17, PFD+18, WBBD15].

Axiomatic models, meanwhile, do not specify systems as state machines. Instead, they specify certain invariants of the system (i.e., things that are always true in the system) that constrain the executions of the model in some way. Axiomatic specifications thus implicitly describe the allowed behaviours of a system by stating the constraints that the system respects. The allowed executions of the model consist of any execution that satisfies these constraints. Axiomatic models defined in the `herd` format (Section 2.3.1) are the de-facto standard for specifying ISA-level MCMs today [AMT14]. Mador-Haim et al. [MHMS+12] also previously defined an axiomatic MCM specification for the Power ISA. PipeCheck [LPM14, LSMB16] (Section 2.4) developed a way to axiomatically specify microarchitectural MCM orderings on which this dissertation builds.

Axiomatic models may be *single-event axiomatic* or *multi-event axiomatic*. Alglave et al. [AMT14] define single-event axiomatic MCM specifications as those that use a single event to denote the propagation of a store operation to other cores/threads. Conversely, they define multi-event axiomatic specifications as those that use multiple events to denote the propagation of a store to other cores/threads. `herd` ISA-level MCM specifications (Section 2.3.1) are examples of single-event axiomatic models. Meanwhile, PipeCheck microarchitectural ordering specifications (Section 2.4) are examples of multi-event axiomatic specifications.

Operational and axiomatic models each have their own advantages and disadvantages. Operational models can be more intuitive than axiomatic ones as they tend to resemble the systems that they model, and they do not require users to come up with invariants of the system to write its model. On the other hand, axiomatic specifications are generally more concise than operational ones [AMT14]. More importantly, as prior work [MHAM10, AKT13, AKNT13, AMT14] shows, the verification performance of axiomatic models can be faster than when using operational models, sometimes by orders of magnitude. To get the best of both worlds, researchers today may define both operational and axiomatic models for the same system and then formally prove the equivalence of the two models [OSS09, MHMS+12, AMT14, PFD+18].

The rest of this chapter covers necessary background and prior work on formal MCM specification and verification. The prior work covered includes research that uses operational models as well as that which uses axiomatic models. The work in this dissertation primarily uses axiomatic models, although the work in Chapter 3 requires the translation of axiomatic properties to RTL properties that are evaluated over an operational model (i.e. the RTL).

2.3 MCM Specification and Verification

This section covers necessary background and related work on MCM specification and verification apart from the PipeCheck work [LPM14, LSMB16], which Section 2.4 discusses in detail.

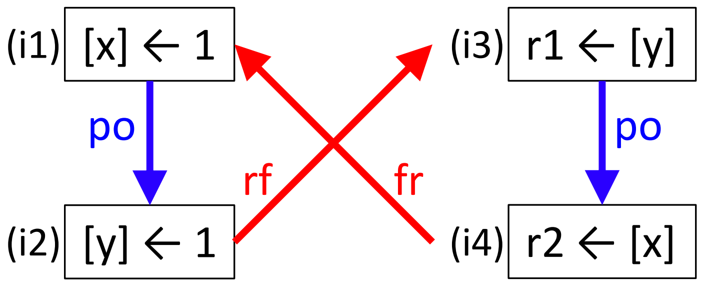


Figure 2.10: ISA-level execution of `mp` forbidden under SC due to the cycle in the *po*, *rf*, and *fr* relations.

2.3.1 Instruction Set (ISA) Memory Consistency Models

The execution of a parallel program on a processor can be modelled at the ISA level (i.e. assembly language level) as a graph, where the nodes are instructions, and edges between nodes represent ISA-level MCM attributes.

ISA-level MCMs can then axiomatically define whether executions are allowed or forbidden based on whether they show certain patterns of relations or not.

Modelling and reasoning about the MCM correctness of executions using the notion of relations between instructions in an execution was first introduced by Shasha and Snir [SS88] for SC. `herd` (developed by Alglave et al. [AMT14]), a widely used framework for specifying and conducting formal analysis of ISA-level MCMs today also models ISA-level MCMs axiomatically using relations. Each individual relation between a pair of instructions in an execution represents some ordering relevant to the MCM.

Figure 2.10 shows the ISA-level execution for the outcome of `mp` from Figure 2.2. The *po* relation represents program order, so $i1 \xrightarrow{po} i2$ and $i3 \xrightarrow{po} i4$ represent that $i1$ is before $i2$ and $i3$ is before $i4$ in program order. The *rf* (reads-from) relation links each store to all loads which read from that store. For example, $i2 \xrightarrow{rf} i3$ represents that $i3$ reads its value from $i2$ in this execution. The *fr* (from-reads) edge between $i4$ and $i1$ enforces that the store that $i4$ reads from comes before the store $i1$ in coherence order (a total order on same-address writes in an execution). The *co* relation (not present in Figure 2.10) is used to represent coherence order (Section A.5). Other ISA-level MCMs require extra relations to model reorderings and fences [AMT14].

In the relational framework of Alglave et al. [AMT14], the permitted behaviours of the ISA-level MCM are defined in terms of the irreflexivity, acyclicity, or emptiness of certain relational patterns. For example, SC can be defined using relational modelling as $acyclic(po \cup co \cup rf \cup fr)$. This means that any execution with a cycle in these four relations is forbidden. The execution in Figure 2.10 contains such a cycle, and so is forbidden under SC, as one would expect for `mp`. `herd` can evaluate a given litmus test against such an axiomatically specified ISA-level MCM and return whether or not a given outcome of the litmus test is allowed or forbidden under that MCM. This dissertation utilises `herd`-style axiomatic specifications for ISA-level MCMs, most notably in Chapter 5. The work in Chapter 4 also makes use of such specifications to decide whether a given litmus test should be allowed or forbidden by hardware.

The past decade or so has seen the creation of formal axiomatic ISA-level MCM specifications for SC, x86-TSO [OSS09], Power [MHMS$^+$12, AMT14], ARMv8 [PFD$^+$18], and RISC-V [RIS19]. Researchers created the formal MCM specifications of TSO, Power, and ARM by discussing the workings of such processors with architects from those companies as well as through dynamic litmus test-based validation (Section 2.3.5) of existing processors implementing those ISAs. Formal ISA-level MCM specifications created through this process closely resemble the behaviour of the processors they model. However, these specifications are not formally verified as being sound with respect to real hardware. This can potentially lead to false negatives, as Section 1.4 covers.

The formal RISC-V MCM specification is somewhat of an exception to the above process, as it was created before a large number of RISC-V chips were taped out. TriCheck [TML$^+$17] was a spur for the creation of this formal specification, as it uncovered severe issues in a draft specification of the RISC-V MCM. This led to an effort (of which I was a part) to create a formal RISC-V MCM specification that fixes these issues. The effort was successful, and the new RISC-V MCM has since been ratified [RIS19].

Other schemes for generating formal MCM specifications also exist. The MemSynth tool [BT17] takes as input a set of litmus test results for a processor and a partially completed MCM specification. It then synthesizes a formal MCM specification for the processor that is guaranteed to respect the litmus test results provided as input. Operational models consisting of abstract machines that model hardware MCM behaviour can also function as MCM specifications, though by their nature they are somewhat intertwined with the hardware implementations that they model. They are covered in Section 2.3.3.

In addition to specifying the MCMs of general-purpose multicore processors, there has been work on formally specifying the MCMs of GPUs [ABD$^+$15, WBSC17, WBBD15, LSG19]. There has also been research on incorporating related hardware features like mixed-size accesses[12] [FSP$^+$17], transactional memory [CSW18] (Section 2.1.2), and virtual memory [RLS10, LSMB16, HTM20] into formal MCM specifications.

2.3.2 Program Verification Under MCMs

As Section 1.3 states, this dissertation focuses on verification of MCM implementations (i.e. hardware and language implementation software like compilers). It does not focus on program verification (the verification of programs against their high-level specifications) under various MCMs. Nevertheless, there has been a considerable amount of work on program verification under various MCMs. Much of this work utilises *stateless model checking*, a variant of model checking that does not explicitly store states in memory [God97]. CDSChecker [ND13] and CDSSpec [OD17] enable

[12]Mixed-size accesses refer to the interaction between multiple memory accesses of different widths, e.g. a 32-bit store and an 8-bit store that overlap in memory.

the exhaustive exploration of all possible behaviours of concurrent data structure code under the C11 memory model, and verification of these data structures against CDSSpec specifications. Nidhugg [AAA$^+$15,AAJL16,AAJ$^+$19] is capable of conducting stateless model checking for assembly language programs under the SC, TSO, PSO, and Power MCMs. RCMC [KLSV17] enables stateless model checking of C/C++ programs under a more recent revision of the C11 memory model [LVK$^+$17]. GenMC [KRV19] is a stateless model checking algorithm that can be used for verifying clients of concurrent libraries, and is parametric in its choice of MCM. Most recently, HMC [KV20] enables stateless model checking of assembly language programs under a wide variety of hardware MCMs, and is faster than Nidhugg.

2.3.3 Hardware Ordering Specifications

The complicated MCM behaviour of commercial processors like those of Power and ARM can be difficult to characterise and understand. To better understand and analyse processors like these, researchers have developed detailed operational models of these processors through dynamic litmus test analysis of the processors (Section 2.3.5) and/or discussion with the architects who created them. These models can evaluate small programs (like litmus tests, but sometimes slightly larger) to determine which of their outcomes should be allowed or forbidden under the processor represented by the model. Some of these models have undergone multiple revisions to incorporate additional detail into the models or to adapt to a revision of the processor's ISA-level MCM. The Power and ARMv7 MCMs are quite similar to each other (Section 2.1.3), and so sometimes the same model (possibly with a small number of changes) is used for both Power and ARMv7.

Sarkar et al. [SSA$^+$11] developed an operational model for the MCM behaviour of Power multiprocessors, and suggested that it might be applicable to ARMv7 with minor changes. Alglave et al. [AMT14] also created an operational model for Power

MCM behaviour, which they proved equivalent to their axiomatic `herd` model for the Power MCM. Gray et al. [GKM$^+$15] extended the model of Sarkar et al. [SSA$^+$11] and integrated it with an ISA model for part of the Power instruction set, while Flur et al. [FGP$^+$16] constructed a similar integrated concurrency and ISA model for ARMv8. The detailed concurrency model of Flur et al. [FGP$^+$16] was known as the "Flowing model". Flur et al. [FSP$^+$17] extended the Flowing model of ARMv8 and the Power model of Gray et al. to handle mixed-size accesses. Most recently, Pulte et al. [PFD$^+$18] developed operational and axiomatic models for ARMv8 that accounted for the ARMv8 MCM's change to implement rMCA write atomicity as opposed to nMCA (Section A.4), and proved the equivalence of these operational and axiomatic models. The operational model of Pulte et al. is a simplification of the Flowing model, and supports mixed-size accesses. Wickerson et al. [WBBD15] developed an operational model of GPU hardware which was capable of implementing the OpenCL programming framework extended with AMD's remote-scope promotion technology [OCY$^+$15]. Their model included the syntax and semantics of a minimal assembly language for their hardware model.

Each of these models is a detailed representation of the behaviours of the processors that it models. However, they are only models of specific types of processors, e.g. an ARMv8 model only represents ARMv8 processors. These models are not part of a generic framework specialised for describing and verifying hardware orderings. This is in contrast to the PipeCheck (Section 2.4) framework, which defines a domain-specific language (μspec) for hardware orderings, and supports the verification of any processor whose orderings are specified in μspec for suites of litmus tests.

2.3.4 Manually Proving MCM Correctness of Hardware Implementations

There are two main frameworks for specifying hardware and formally verifying its MCM guarantees: PipeCheck [LPM14, LSMB16] and Kami [VCAD15, CVS^{+}17]. This section describes Kami, while Section 2.4 describes PipeCheck.

Kami enables hardware designers to write their hardware in the Coq proof assistant, prove its correctness, and then extract a hardware implementation from this proven-correct hardware written in Coq. In the first paper on what became Kami, Vijayaraghavan et al. [VCAD15] formalised the idea of components in a hardware design such as reorder buffers (ROB), register files, store buffers, and caches as labelled transition systems (LTSes). They provided a machine-checked proof in Coq that if the models of the individual components obey certain rules, then the combined system implements sequential consistency for all programs. Kami thus enabled the modular MCM verification of models of hardware.

The Bluespec hardware description language enables users to write hardware as a set of rules, and the hardware then behaves as if each of the rules were executed atomically, with a scheduler automatically choosing the order in which they are executed. LTS semantics are quite close to those of the Bluespec, and in some cases can be transliterated directly to Bluespec. However, the work of Vijayaraghavan et al. [VCAD15] only proved correctness of a model of the processor rather than its actual implementation. It also lacked an automated process to extract a Bluespec hardware implementation from its proven-correct model in Coq. More recently, Choi et al. [CVS^{+}17] extended the work of Vijayaraghavan et al. to create Kami. Kami enabled users to modularly prove the correctness of hardware itself, rather than a model. Kami also included an automated process for extracting a Bluespec hardware implementation from its proven-correct Coq model.

While Kami can prove hardware MCM correctness across all programs, its proofs require formal methods expertise and significant manual effort. This is because Kami requires such proofs to be written in the Coq proof assistant. Kami also can currently only prove correct hardware written in Bluespec's rule-based style, rather than Verilog (which is much more common). Furthermore, Kami has thus far only proven that hardware designs implement SC [VCAD15, CVS$^+$17], rather than demonstrating how to handle weak MCMs in such a framework.

The work in this dissertation is capable of automatically verifying the MCM correctness of hardware RTL for litmus tests (Chapter 3), and is also able to automatically prove the MCM correctness of hardware designs across all possible programs (Chapter 5). Furthermore, Chapter 4 shows how to automatically conduct modular bounded MCM verification of hardware designs. This dissertation thus shows that the use of automated verification does not preclude the sorts of guarantees that Kami provides in its proofs.

2.3.5 Dynamic MCM Verification

Dynamic MCM verification observes the behaviour of a processor as it runs and flags MCM violations that occur. Dynamic verification using litmus tests consists of running litmus tests on a parallel system and checking if any of the litmus test's outcomes that are forbidden by the system's MCM are observable. To increase the probability of bug discovery, a large number of different tests can be used, the tests can be run a large number of times, and various techniques can be used to stress the parallel system in order to expose weak (i.e. non-SC) behaviours [AMSS11, SD16]. Dynamic verification may also use a runtime approach, where it tracks all instructions as the processor executes them and reports any MCM violations that occur [MS09, CMP08].

TSOTool [HVML04] runs psuedo-randomly generated programs with conflicting accesses on a processor that claims to implement TSO, and compares their results to TSO requirements. The algorithm checking results against TSO requirements runs in polynomial time. `litmus` [AMSS11] runs litmus tests for many iterations against hardware to find interesting behaviour. It includes techniques to make such behaviour appear more frequently. DVMC [MS09] conducts dynamic MCM verification at runtime. It tracks instructions as the processor executes them, and checks that the processor maintains invariants sufficient to implement the MCM for the executed instructions, reporting any invariant violations that occur. DVMC has the benefit of also being able to detect physical (circuit-level) errors that lead to MCM failures, and supports the SC, TSO, PSO, and RMO consistency models (Section 2.1.3). Chen et al. [CMP08] also conduct runtime dynamic MCM verification, but verify MCM behaviour through analysis of a constraint graph, which reduces the number of false positives when compared to DVMC. Like DVMC, their work can also detect physical errors that lead to MCM failures.

Nevertheless, dynamic verification can never guarantee the absence of MCM bugs in hardware. This is because it only verifies executions actually observed on a processor (if using runtime verification) or observed executions of litmus tests (if dynamically running litmus tests). There is always the chance that the system contains an MCM bug that does not occur in the observed executions, limiting the correctness guarantees that such tools can provide. Furthermore, such approaches require a runnable implementation to exist in order to work, in contrast to approaches like Kami (Section 2.3.4) or PipeCheck (Section 2.4).

2.4 Automated Formal Microarchitectural MCM Verification with PipeCheck

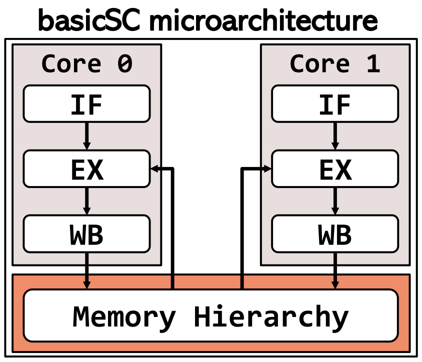
A large amount of prior MCM verification only verified hardware using dynamic approaches or simply assumed that hardware was correct. There was no way for computer architects and hardware engineers to formally verify their hardware designs against ISA-level MCMs for correctness. PipeCheck [LPM14, LSMB16] filled this verification gap by enabling the automated formal verification of microarchitectural ordering specifications against their ISA-level MCMs for suites of litmus tests. Notably, PipeCheck's verification can be conducted long before RTL is written. The automated nature of PipeCheck's verification makes it amenable to use by computer architects and hardware engineers. This is in contrast to tools like Kami (Section 2.3.4), which are difficult for individuals without formal methods expertise to use.

PipeCheck involves three main components: the μhb (microarchitectural happens-before) graph formalism for modelling microarchitectural executions, the μspec domain-specific language for specifying a design's microarchitectural orderings, and an algorithm that uses μhb graphs for model checking the executions of a litmus test on a design whose orderings are specified in μspec.[13] This section explains each of these in turn, followed by a summary of PipeCheck's deficiencies (each of which are addressed by this dissertation).

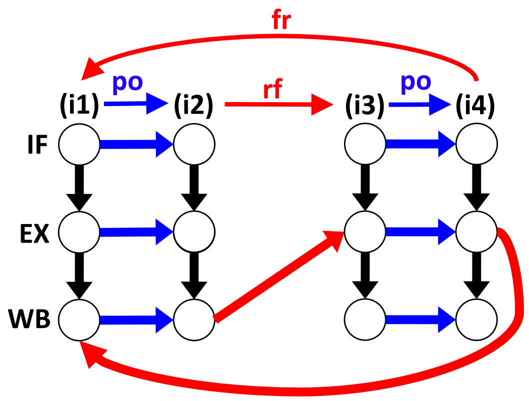
2.4.1 Microarchitectural Happens-Before (μhb) Graphs

PipeCheck's verification requires it to model and reason about the ordering of events in the executions of litmus tests on hardware designs. To do so, PipeCheck developed a formalism of μhb (microarchitectural happens-before) graphs, where a given μhb graph represents a microarchitectural execution. Nodes in these graphs represent microarchitectural events in the execution of an instruction. These can represent events such as an instruction reaching a particular pipeline stage, as well as other types of microarchitectural events. CCICheck [MLPM15] subsequently added the

[13]Recall that in this dissertation, the term "PipeCheck" refers to the current incarnation of the PipeCheck approach as realised in its follow-on work COATCheck [LSMB16]. Of PipeCheck's three components, the μhb graph formalism was developed in the initial PipeCheck paper [LPM14], while the μspec domain-specific language and the model checking algorithm described in this section were developed as part of COATCheck.



(a) The `basicSC` microarchitecture, where both cores have 3-stage in-order pipelines of Fetch (`IF`), Execute (`EX`), and Write-back (`WB`) stages.



(b) μhb graph for `mp`'s forbidden outcome on `basicSC`.

Figure 2.11: Example μhb graph for the `mp` litmus test and example μspec axiom.

capability to model cache occupancy and coherence protocol events in μhb graphs through its ViCL (Value in Cache Lifetime) abstraction.

Edges in μhb graphs represent happens-before relationships between nodes. So an edge between nodes A and B in a μhb graph would indicate that event A happens before event B in the execution modelled by the μhb graph. The edge does not make any claims on the duration of time between A and B, only that A happens before B in the execution. The set of edges in a μhb graph is closed under transitivity. In other words, if a μhb graph contains edges from A to B and from B to C, then it also includes an edge from A to C by transitivity. μhb edges implied by transitivity are not shown in the μhb graphs in this dissertation for the sake of readability.

Figure 2.11b shows an example μhb graph for the execution of `mp`'s non-SC outcome (`r1=1,r2=0`) on the microarchitecture of Figure 2.11a (henceforth called `basicSC`), which aims to implement SC. Above the μhb graph is the corresponding ISA-level cycle in $po \cup co \cup rf \cup fr$ (also seen in Figure 2.10), showing that SC requires this execution to be forbidden.

Each column in the μhb graph represents an instruction flowing through the pipeline, and each node represents a particular event in the execution of an instruction.

In `basicSC`, each core has three-stage in-order pipelines of `Fetch` (`IF`), `Execute` (`EX`), and `Writeback` (`WB`) stages. There are thus three μhb nodes for each instruction, denoting when it reaches the `IF`, `EX`, and `WB` stages.[14] For instance, the leftmost node in the second row represents instruction `i1` at its `Execute` stage, while the second node in the second row represents instruction `i2` at its `Execute` stage. The blue edge between these two nodes enforces that they pass through the `Execute` stage in order, as required by the in-order pipeline. Other blue edges between the `Fetch` stages and between the `Writeback` stages of instructions on the same core similarly enforce the in-order nature of `basicSC`'s other pipeline stages. The colouring of edges is purely for readability purposes; all μhb edges are of equal strength for PipeCheck's formal analysis. This is in contrast to ISA-level MCM specifications (Section 2.3.1) where edges are treated differently depending on their type.

Other edges in Figure 2.11b's μhb graph represent other ordering relationships that the designer stipulates for the implementation. For example, in order for `i4` to return a value of 0 for its load of `x`, it must complete its Execute stage before the store of `x` in `i1` reaches its Writeback stage and goes to memory, thus overwriting the old value of 0 for `x`. This ordering is represented by the red edge from `i4`'s `EX` node to `i1`'s `WB` node in Figure 2.11b. Likewise, in order for `i3`'s load of `y` to read a value of 1 in its `EX` stage, it must occur after the store of `y` in `i2` reaches the `WB` stage. This ordering is shown by a red edge between those two nodes.

If a μhb graph contains a cycle, then for any node D that is part of that cycle, the transitivity of μhb edges implies that D must happen before itself. This is impossible, and thus a cyclic μhb graph represents an execution that is *unobservable* on the target microarchitecture. Likewise, an acyclic μhb graph represents an execution that is *observable* on the target microarchitecture. A topological sort of an acyclic μhb graph

[14]Note that the organisation of the μhb graph into rows and columns is purely for readability purposes; the only events and orderings modelled by a μhb graph are those denoted by its nodes and edges.

```
Axiom "IF_FIFO":
forall microops "a1", "a2",
(SameCore a1 a2 /\ ~SameMicroop a1 a2) =>
  EdgeExists((a1,Fetch), (a2,Fetch)) =>
    EdgeExists((a1,Execute), (a2,Execute)).
```

Figure 2.12: μspec axiom expressing that the Fetch pipeline stage should be FIFO on `basicSC`.

is a trace of an execution modelled by that μhb graph. The graph in Figure 2.11b is cyclic (the cycle is comprised of the two red edges and the blue edges that connect them), so this microarchitectural execution is unobservable, as SC requires for `mp`.

2.4.2 The μspec Domain-Specific Language

PipeCheck developed the μspec domain-specific language to specify the orderings enforced by a given microarchitecture. A μspec specification of a microarchitecture allows PipeCheck to decide which nodes and edges must be present in that microarchitecture's μhb graphs. A μspec specification consists of a set of *axioms*, each of which specifies a property that must hold for every μhb graph on that microarchitecture.[15] These axioms each correspond to one of the smaller orderings enforced by the microarchitecture, so a set of μspec axioms constitutes a design description of microarchitectural orderings. PipeCheck verifies whether the combination of these individual axioms is enough to enforce the system's MCM for a given litmus test.

Figure 2.12 shows an example μspec axiom which enforces that the `Fetch` pipeline stage is FIFO on `basicSC`. The axiom applies to all pairs of instructions `a1` and `a2` in the litmus test being verified that are on the same core (`SameCore a1 a2`) where `a1` and `a2` are distinct instructions (`~SameMicroop a1 a2`). For such pairs of instructions, if an edge exists between their `Fetch` stages (as denoted by the first `EdgeExists` predicate), then an edge must also exist between their `Execute` stages

[15]A μspec axiom can thus be thought of as an invariant for the ordering behaviour of the microarchitecture.

```
⟨microarch_spec⟩      ::= ⟨axiom_list⟩

⟨axiom_list⟩          ::= ⟨axiom_list⟩ ⟨axiom⟩
                        |  ⟨axiom⟩

⟨axiom⟩               ::= ⟨formula⟩

⟨formula⟩             ::= 'forall microop' ⟨microop_id⟩',' ⟨formula⟩
                        |  'exists microop' ⟨microop_id⟩',' ⟨formula⟩
                        |  ⟨formula⟩ ∧ ⟨formula⟩
                        |  formula ∨ formula
                        |  ∼ formula
                        |  ⟨predicate⟩

⟨predicate⟩           ::= ⟨microop_predicate⟩
                        |  ⟨graph_predicate⟩

⟨graph_predicate⟩     ::= 'NodeExists'
                        |  'NodesExist'
                        |  'EdgeExists'
                        |  'EdgesExist'
```

Figure 2.13: The core grammar of the μspec domain-specific language for specifying microarchitectural orderings [Lus15]. A *microop* is a single load, store, or synchronization (e.g., fence) operation. ⟨*microop_predicate*⟩ refers to a predicate whose result is the same for every μhb graph for a given litmus test and microarchitecture, like `SameCore a1 a2`. ⟨*graph_predicate*⟩ refers to a predicate whose value may change depending on which μhb graph of a litmus test and microarchitecture it is evaluated for.

(signified by the second `EdgeExists` predicate) to satisfy the axiom. In the case of the μhb graph in Figure 2.11b for `mp` on `basicSC`, `i1` and `i2` constitute a pair of distinct instructions on the same core with an edge between their `Fetch` stages. The axiom thus adds an edge between their `Execute` stages to make the `EdgeExists` predicate which refers to the `Execute` stage true. Instructions `i3` and `i4` also satisfy the axiom's conditions on `a1` and `a2`, and so an edge is added between their `Execute` stages as well.

The μspec language is based on propositional logic but includes support for quantifiers over instructions and the transitivity of μhb edges. Figure 2.13 details the core grammar of the μspec language. The building blocks of μspec are its built-in predicates, which can be divided into two types. Graph predicates are predicates that reason about the nodes and edges in μhb graphs ($\langle$*graph_predicate*$\rangle$ in Figure 2.13). Meanwhile, microop predicates are predicates that reason about the microops (individual loads, stores, or synchronization instructions like fences) whose events are present in the μhb graph ($\langle$*microop_predicate*$\rangle$ in Figure 2.13). μspec's graph predicates are `NodeExists`, `NodesExist`, `EdgeExists`, and `EdgesExist`, which correspondingly take μhb nodes and edges as parameters. Meanwhile, an example of a μspec microop predicate is `SameCore a1 a2` from Figure 2.12, which returns true if and only if the microops `a1` and `a2` are on the same core for the program being modelled.

The value of a graph predicate may change depending on what nodes and edges are present in a μhb graph, but the value of a microop predicate will be the same for every μhb graph for a particular microarchitecture and litmus test. For instance, if evaluating Figure 2.12's `SameCore a1 a2` predicate where `i` and `j` are `i1` and `i2` from `mp`, this predicate will always be true no matter what nodes and edges are in the μhb graph it is evaluated for. On the other hand, if evaluating the `EdgeExists((a1,Execute), (a2,Execute))` predicate from Figure 2.12, it will return true or false depending on whether or not the edge between the `Execute` stages exists in the μhb graph that the predicate is evaluated for.

Lustig's dissertation [Lus15] provides further details on μspec.

2.4.3 Automatically Verifying Correctness of a Litmus Test

There are usually multiple μhb graphs for a given litmus test outcome and μspec specification, each corresponding to different microarchitectural executions and event orderings. For instance, if modelling executions of `mp`'s non-SC outcome (`r1=1,r2=0`) on a typical processor, the load `i3` may read its value from the cache in one execution

Test Outcome	**$\geq$ 1 Acyclic Graph (Observable)**	**No Acyclic Graphs (Unobservable)**
Allowed	OK	OK (Strict)
Forbidden	**MCM violation**	OK

Table 2.2: Summary of how PipeCheck interprets SMT solver output to verify a given litmus test.

and from main memory in another execution. The μhb graphs for these two executions would be different.

For a given litmus test outcome to be observable on a microarchitecture, there must exist an acyclic μhb graph for that litmus test outcome which satisfies all the μspec axioms in the microarchitecture's ordering specification. A search for such an acyclic μhb graph can be converted into a query to an SMT solver as outlined here. First, each μspec axiom is instantiated for the litmus test being verified. This is done by grounding μspec's `forall` and `exists` quantifiers. `forall` quantifiers are turned into ANDs over the litmus test instructions, and `exists` quantifiers are turned into ORs over the litmus test instructions. For example, if verifying `mp`, the quantifier `forall microop "j", <formula>` would be translated as follows. Four instances of `<formula>` would be ANDed together, and in each instance of `<formula>` `j` would be replaced with one of `i1`, `i2`, `i3`, and `i4` from `mp`, with each instruction being used for only one instance of `<formula>`.

Next, all microop predicates are evaluated according to the litmus test outcome being verified. Since the value of a microop predicate for a particular litmus test outcome is independent of the μhb graph it is evaluated over, they can be directly evaluated to true or false before the query is passed to an SMT solver. Once this is done, each simplified axiom consists solely of graph predicates combined with AND ($\wedge$), OR ($\vee$), and NOT ($\sim$). The combination of these axioms is then passed to the SMT solver as a formula to solve.

An assignment to the graph predicates in the formula passed to the SMT solver describes a μhb graph. For instance, if the predicate `EdgeExists((a1,Fetch), (a2,Fetch))` is true in the assignment, then that edge exists in the graph that the assignment describes. Likewise, if the predicate is false, the edge does not exist in the graph. If an assignment to the graph predicates exists which satisfies all the axioms (i.e. a satisfying assignment), and the μhb graph described by that assignment is acyclic, then the litmus test outcome is observable on the microarchitecture. If the formula passed to the solver is unsatisfiable without making the μhb graph cyclic (i.e. unsatisfiable modulo the transitivity of μhb edges), then the litmus test outcome is guaranteed to be unobservable on the microarchitecture.

PipeCheck uses a custom SMT solver to search for an acyclic μhb graph to verify whether a microarchitecture correctly respects its MCM for a given litmus test. Table 2.2 covers the four possible cases for PipeCheck's litmus test verification. If a litmus test outcome is forbidden, and PipeCheck's solver cannot find an acyclic μhb graph satisfying the microarchitecture's μspec axioms for that outcome, then the outcome is unobservable on the microarchitecture as required. On the other hand, if PipeCheck's solver does find an acyclic μhb graph satisfying the microarchitecture's μspec axioms for a forbidden outcome, then the microarchitecture is buggy. The acyclic μhb graph found by PipeCheck is provided to the user as a counterexample for that litmus test outcome.

If a litmus test is allowed, the user would expect PipeCheck to find an acyclic μhb graph indicating that the outcome is microarchitecturally observable. Even if no acyclic μhb graph can be found for an allowed test, it just means that the microarchitecture is overly strict and forbidding more behaviours than its MCM deems necessary.

PipeCheck uses a custom SMT solver written in Gallina (the functional programming language of Coq), the language in which PipeCheck is implemented. It explicitly keeps track of the μhb graph represented by variable assignments that the solver is considering so as to check the graph's cyclicity. Standard SMT solvers like Z3 [dMB08] are also capable of conducting the analysis done by PipeCheck's solver. The Linear Integer Arithmetic (LIA) theory of such SMT solvers can be used to check graph cyclicity. This is a well-known technique and is used by the RealityCheck work from Chapter 4, as Section 4.6.4 describes.

2.4.4 Moving Beyond PipeCheck

PipeCheck was a large stride forward in the field of hardware MCM verification. Nevertheless, as stated in Section 1.4 (and summarised below), PipeCheck has a number of shortcomings that limit its effectiveness in verifying real-world processors. Firstly, PipeCheck provides no way to link its verification to that of RTL. This prevents verification of the soundness of a μspec specification with respect to pre-existing RTL (if RTL was written first). It also prevents verification that RTL matches a pre-existing μspec specification (if the μspec was written first, as recommended for progressive verification (Chapter 6)). Secondly, PipeCheck verifies a microarchitecture for a litmus test all at once. This is intuitive because enforcing the MCM is a responsibility of the entire microarchitecture. However, it also means that once the design gets past a certain level of detail, PipeCheck's verification will not scale due to the NP-completeness of the SMT solver it uses. Thirdly, a hardware design must respect its MCM across all possible programs to ensure correct parallel system operation. However, PipeCheck can only verify a single litmus test at a time. The question of whether verifying the MCM correctness of a microarchitecture across all programs can be reduced to verifying its MCM correctness for a particular suite of litmus tests is still an open problem. As a result, PipeCheck cannot guarantee microarchitectural MCM correctness across all programs.

This dissertation addresses each of the above challenges. RTLCheck (Chapter 3) addresses the issue of verifying μspec model soundness and linking to RTL verification.

RealityCheck (Chapter 4) addresses the problem of scalable microarchitectural MCM verification. PipeProof (Chapter 5) addresses the problem of conducting microarchitectural MCM verification across all possible programs.

2.5 Summary

Common hardware and software optimizations that were developed in the single-core era result in violations of sequential consistency in multicore systems. Rather than forgo these optimizations, hardware and software manufacturers have embraced weak/relaxed MCMs that allow them. (Some hardware optimizations may also be implemented speculatively.) These weak MCMs introduce substantial additional system complexity, both for their specification and for the verification of their implementations.

In response to this complexity, there has been a large amount of work on MCMs and their verification, especially over the past decade. While much of this work is extremely valuable, a number of unresolved challenges in hardware MCM verification remain unaddressed by prior work. Firstly, prior ISA-level MCM specifications and PipeCheck microarchitectural models are not formally verified as being sound with respect to the real hardware that they model. This is problematic because verification using unsound models can miss bugs (i.e., false negatives). Secondly, the monolithic MCM verification conducted by approaches like PipeCheck cannot scale to detailed models of large designs, such as those of commercial processors. This is because the SMT solvers used by such approaches are NP-complete. Thirdly, no prior automated approach provides hardware MCM verification across all programs. This is problematic because hardware MCM verification must cover all possible programs to ensure correctness. However, the only instances of prior work capable of proving hardware MCM correctness across all programs are manual approaches like Kami. This dissertation addresses each of these challenges, beginning with the issue of soundness verification in Chapter 3.

Chapter 3

Checking Soundness and Linking to RTL Verification[1]

Fools dwelling in darkness, but thinking themselves wise and erudite, go round and round, by various tortuous paths, like the blind led by the blind.
—Katha Upanishad
(tr. Swami Nikhilananda)

MCM verification is critical to parallel system correctness, including that of parallel hardware. In response, prior work developed formal specification frameworks for ISA-level MCMs (`herd`) and microarchitectural orderings (μspec), as well as model checking algorithms for verifying the correctness of such specifications for suites of litmus tests. However, these specifications were never formally linked to processor implementations written in RTL like Verilog. This has two disadvantages. Firstly, if creating a formal specification for an existing processor, it prevents engineers from verifying that the specification is a *sound* representation of the underlying hardware. Secondly, if creating a formal specification for a processor whose RTL does not yet exist, the lack of such linkage prevents the verification of the eventual RTL for compliance with the formal specification.

[1]An earlier version of the work in this chapter was previously published and presented by myself at the MICRO-50 conference [MLMP17]. I was the first author on the publication.

Both of these verification tasks require a method for translating formal hardware specifications to equivalent RTL properties. To this end, this chapter presents RTLCheck, a methodology and tool for the automatic translation (given appropriate mapping functions) of μspec axioms to equivalent SystemVerilog Assertions [IEE13] (a well-known language for formally specifying RTL properties) for suites of litmus tests. These SystemVerilog assertions can be formally proven on an RTL implementation by commercial tools such as Cadence JasperGold [Cad15b]. This chapter details RTLCheck's methodology, including the technical challenges involved in the translation, and presents quantitative results from evaluating RTLCheck on the Multi-V-scale processor, a multicore version of the open-source RISC-V V-scale processor [RIS15].

3.1 Introduction

Verifying that hardware correctly implements its MCM is increasingly critical given its fundamental importance (Section 1.3). System verification is a general challenge, with verification costs now dominating total hardware design cost [Fos15]. MCM verification is particularly challenging since it requires analysis across many possible interleavings of events.

In response to the need for formal hardware MCM analysis and verification, prior work developed the `herd` framework [AMT14] for formal analysis and specification of ISA-level MCMs (Section 2.3.1) and the PipeCheck framework [LPM14, LSMB16] for specification and litmus test-based verification of microarchitectural orderings against ISA-level MCMs (Section 2.4). Both `herd` and PipeCheck significantly improved the state of the art in hardware MCM analysis and verification, but neither of them was formally linked to real processor implementations written in RTL like Verilog. `herd` specifications are only validated against real hardware through dynamic testing and consultation with architects. Similarly, PipeCheck specifications are written based on the user's understanding of the microarchitecture being modelled.

This disconnection of formal hardware ordering specifications from RTL makes it difficult to ensure the MCM correctness of real-world processors. The precise nature of the difficulty depends on whether the specification is written before or after the RTL. This chapter focuses on connecting PipeCheck μspec models to RTL, but in combination with PipeProof (Chapter 5), it also accomplishes the formal linkage of `herd` ISA-level MCM specifications to RTL.

If creating a μspec model for an existing processor, the RTL exists prior to the μspec. In this case, there is a need to verify that the μspec model is a *sound* representation of the processor. If a model of the processor is sound, then a proof of the model's correctness implies the correctness of the actual processor. An unsound model will allow a correctness proof to be completed even if the real processor is buggy (i.e., false negatives), which will lead to bugs slipping through PipeCheck's formal verification. As such, verifying the soundness of PipeCheck μspec models is critical to the validity of PipeCheck's verification.

Meanwhile, if creating a μspec model for a future processor, the μspec exists prior to the RTL. In this case, the design can be verified for MCM correctness before RTL is written.[2] This verification can be for litmus tests with PipeCheck, or for all programs with the PipeProof work from this dissertation (Chapter 5). While such verification can ensure the correctness of the design, there remains a need in this flow to verify that the eventual processor RTL also respects the MCM of its ISA. A hardware design may be verified as correctly implementing its ISA-level MCM, but if the RTL written to implement that design does not ensure the design's specified orderings, the taped-out chip will still have MCM bugs. PipeCheck's and PipeProof's verification ensures that the combination of the individual axioms in a μspec model satisfies the ISA-level MCM

[2]Such pre-RTL verification is an important part of the Progressive Automated Formal Verification flow (Chapter 6).

for litmus tests and all programs respectively. This enables the MCM verification of the processor RTL to be reduced to the verification that the RTL satisfies each of the individual μspec axioms.

Both the verification of μspec axiom soundness and the verification that RTL satisfies μspec axioms require a way to check that μspec axioms truly hold on a processor's RTL implementation. This in turn requires a method to translate μspec axioms to equivalent RTL properties that can be verified against an RTL implementation. There has been a large amount of work on formal RTL verification [EF18], notably using languages like Property Specification Language (PSL) [IEE10] and more recently, SystemVerilog Assertions (SVA) [IEE13]. SVA is based on Linear Temporal Logic (LTL), a logic for specifying properties of reactive systems [BK08]. LTL is highly amenable to model checking and has been used to verify systems for decades [PP18]. Similarly, SVA assertions can be formally verified against an RTL implementation using commercial tools such as Cadence JasperGold [Cad15b]. However, prior work in this area has crucially not looked at the verification of multicore MCM properties.

Ideally, if translating μspec axioms to RTL properties, the RTL properties used should be SVA assertions. This has multiple advantages, including easier integration with existing industry practices (given the prevalence of SVA) and the ability to leverage all the advances in SVA and RTL verification over the past decades. Verification of the translated assertions using a tool like JasperGold would then be equivalent to verifying that the μspec axioms hold on the RTL implementation.

Translating μspec axioms to SVA is non-trivial for two major reasons. Firstly, the logics used for μspec and SVA—as well as their semantics—are starkly different from each other. In particular, μspec models are axiomatic while SVA assertions are evaluated over an operational model of RTL. Secondly, SVA verifiers like Cadence JasperGold do not faithfully implement the complete SVA specification. They use an over-approximation[3] when conducting verification that can provide better performance for certain types of properties, but introduces false positives while doing so. If a translation procedure does not take care to work around the false positives of this over-approximation, then JasperGold will find purported violations of its generated SVA assertions that do not correspond to real bugs in the RTL implementation, making it very hard (if not impossible) to determine the correctness of the RTL.

To enable the linkage of μspec axioms to RTL, this chapter presents RTLCheck, a methodology and tool for automatically translating μspec axioms to SVA assertions for a given litmus test. RTLCheck's translation procedure successfully bridges the gap between the disparate logics and semantics at microarchitecture and RTL, while also working around the aforementioned over-approximation used by SVA verifiers. Given a μspec microarchitectural specification, an RTL implementation, and a mapping between microarchitectural primitives (e.g. individual μspec nodes) and their corresponding RTL signals and values, RTLCheck automatically generates SystemVerilog Assertions (on a per-test basis) and adds them to the RTL implementation. RTLCheck then uses JasperGold to check these assertions on the RTL. The results of this verification say whether the asserted properties have been proven for a given test, whether they have been proven for the test up to a bounded number of cycles, or if a counterexample (an execution trace that does not satisfy the property) has been found. If a counterexample is found, a discrepancy exists between the microarchitectural specification and RTL. This corresponds to a bug in the μspec model (if verifying soundness) or a bug in the RTL (if verifying that RTL respects the μspec axioms).

As a case study, this chapter demonstrates RTLCheck's usage on a multicore version of the RISC-V V-scale open-source processor design[4] [RIS15]. In doing so,

[3]An over-approximation attempts to prove a property stronger than the one the user requested, or uses a system specification that is weaker than the actual system specification provided by the user. It is a conservative approximation, so while it may give false positives, it will not give false negatives.

[4]V-scale has been deprecated since the research in this chapter was initially conducted [Mag16], but it remains an interesting case study.

RTLCheck discovers a bug in the V-scale processor's memory implementation. After fixing the bug, I use RTLCheck to show that the multicore V-scale RTL satisfies a set of microarchitectural axioms that are sufficient to guarantee SC for 56 litmus tests. JasperGold discovers complete proofs (i.e. true for all possible traces of a given litmus test) for 89% of the generated SVA properties in 11 hours of runtime, and can generate bounded proofs (i.e. true for all possible test traces up to a certain number of cycles) for the remaining properties.

RTLCheck's linkage of μspec axioms to RTL enables (on a per-test basis) the automated soundness verification of μspec models with respect to RTL, as well as the automated verification that RTL satisfies μspec axioms. Such verification helps push the correctness guarantees of early-stage MCM verification tools like PipeCheck (Section 2.4) and PipeProof (Chapter 5) down to real processor implementations, improving the correctness of taped-out chips. It also fills the requirement for post-implementation verification in a progressive verification flow (Section 6.5) for MCM properties in parallel processors. RTLCheck's test-specific assertion generation also serves as a stepping stone towards the translation of μspec axioms to SVA assertions that are general across all programs.

The remainder of this chapter is organised as follows. Section 3.2 provides a motivating example for RTLCheck's use. Section 3.3 provides a high-level overview of RTLCheck. Section 3.4 provides necessary background on the syntax and semantics of SVA assertions, which are the type of assertions generated by RTLCheck. Section 3.5 explains the over-approximation used by SVA verifiers like JasperGold, the challenge it introduces when conducting MCM verification, and RTLCheck's solution to this challenge. Section 3.6 covers the details of RTLCheck's procedure for translating μspec axioms to SVA assertions and assumptions for a given litmus test. Section 3.7 briefly explains the microarchitecture and corresponding μspec model of the Multi-V-scale processor which serves as this chapter's case study. Section 3.8 covers RTLCheck's

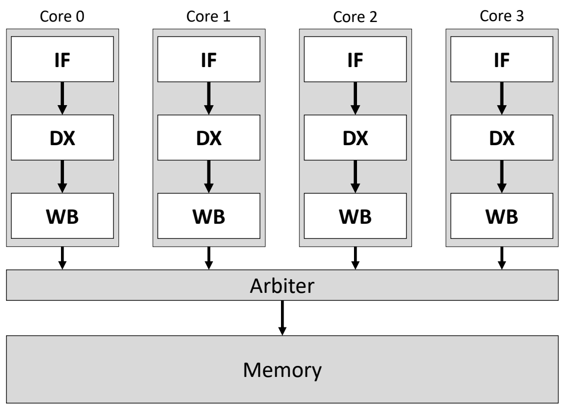


Figure 3.1: The Multi-V-scale processor: a simple multicore processor with four three-stage in-order pipelines. The arbiter allows only one core to access memory at a time.

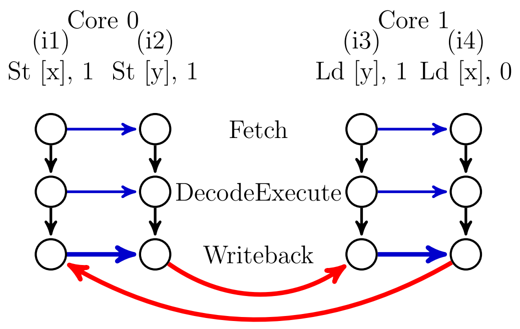
Core 0	**Core 1**
(i1) [x] ← 1	(i3) r1 ← [y]
(i2) [y] ← 1	(i4) r2 ← [x]
Under SC: Forbid r1=1, r2=0	

Figure 3.2: Code for litmus test `mp`

experimental methodology. Section 3.9 covers RTLCheck's results, including a bug it found in the V-scale processor. Section 3.10 covers related work specifically related to RTLCheck, and Section 3.11 summarises the chapter.

3.2 Motivating Example

Figure 3.1 shows the Multi-V-scale processor, a simple multicore where each core has a three-stage in-order pipeline. Instructions in these pipelines first go through the Fetch (IF) stage, then a combined Decode-Execute (DX) stage, and finally a Writeback (WB) stage where data is returned from memory (for loads) or sent to memory (by stores). An arbiter enforces that only one core can access data memory at any time. The read-only instruction memory (not shown) is concurrently accessed by all cores. This processor is simple enough that it appears to implement sequential consistency (SC), but how can one formally verify that its RTL indeed does so?



(a) μhb graph for the SC-forbidden outcome of Figure 3.2's `mp` litmus test on Figure 3.1's processor. The cycle in this graph (shown by the bolded edges) shows that this scenario is correctly unobservable at the microarchitecture level.

```
Axiom "WB_FIFO":
forall microops "a1", "a2",
(OnCore c a1 /\ OnCore c a2 /\
  ~SameMicroop a1 a2 /\ ProgramOrder a1 a2) =>
EdgeExists((a1,DX)), (a2,DX))) =>
AddEdge((a1,WB)), (a2,WB))).
```

(b) Axiom expressing that the WB stage should be FIFO.

```
always @(posedge clk) begin
  if (reset | (stall_DX & ~stall_WB)) begin
     // Pipeline bubble
     PC_WB <= `XPR_LEN'b0;
     store_data_WB <= `XPR_LEN'b0;
     alu_out_WB <= `XPR_LEN'b0;
  end else if (~stall_WB) begin
     //Update WB pipeline registers
     PC_WB <= PC_DX;
     store_data_WB <= rs2_data_bypassed;
     alu_out_WB <= alu_out;
     csr_rdata_WB <= csr_rdata;
     dmem_type_WB <= dmem_type;
  end
end
```

(c) Verilog RTL responsible for updating WB pipeline registers.

Figure 3.3: Illustration of the MCM verification gap between microarchitectural axioms and underlying RTL. The axiom in (b) states that the processor WB stage should be FIFO. Sets of such axioms can be used to enumerate families of μhb graphs such as the one in (a). RTLCheck translates axioms such as (b) to equivalent SVA assertions at RTL on a per-test basis. These properties can be verified to ensure that Verilog such as that in (c) upholds the microarchitectural axioms for a given test.

Even for a design as small as Multi-V-scale, verifying the MCM correctness of its RTL is non-trivial. The actual RTL of Multi-V-scale is substantially more detailed than Figure 3.1. Consider Figure 3.3c, which shows the portion of Multi-V-scale's Verilog RTL that updates the WB pipeline registers from the DX pipeline registers. Processor RTL is very low-level, and trying to verify the MCM correctness of the entire processor as a single verification problem is quite difficult. This verification can be broken down into two pieces, along the layers of the hardware-software stack. The first is to verify a microarchitecture-level ordering model for correctness against the ISA-level MCM. The second is to verify that RTL respects the orderings of the microarchitectural model.

PipeCheck [LPM14, LSMB16] (Section 2.4) developed methods for conducting microarchitectural MCM verification against an ISA-level MCM for suites of litmus tests by model checking μspec microarchitectural ordering specifications using microarchitectural happens-before (μhb) graphs. Figure 3.3a shows an example PipeCheck μhb graph for the `mp` litmus test[5] (Figure 3.2) running on two cores of a μspec model of the Multi-V-scale processor. The graph contains a cycle (comprised of the four thicker red and blue edges), indicating that the execution it represents is unobservable on the μspec model of Multi-V-scale. Ordering rules specifying when and where to add edges to a μhb graph are described in terms of μspec axioms such as the one in Figure 3.3b. This axiom states that if the DX stage of an instruction `a1` happens before the DX stage of an instruction `a2` that is later in program order on the same core, then the WB stage of `a1` must also happen before the WB stage of `a2`. This `WB_FIFO` axiom is responsible for the μhb edges between the WB stages of `i1` and `i2` and those of `i3` and `i4` in Figure 3.3a.

The microarchitectural verification conducted by tools such as PipeCheck is only valid if each of the individual ordering axioms is actually upheld by the underlying

[5]`mp` was explained in Section 2.1.1.

RTL. Thus, there is a need to verify that these axioms *are* respected by the RTL of the design. If they are not, then any microarchitectural verification assumes incorrect orderings and is invalid. For example, if Figure 3.3b's axiom from the Multi-V-scale model was not actually maintained by the Multi-V-scale RTL in some execution of the `mp` litmus test, then the assumed happens-before edges between the WB stages of instructions on the same core would not actually exist. Without them, there is no cycle in the graph, resulting in the forbidden outcome becoming observable.

The key contribution of RTLCheck lies in verifying (on a per-test basis) that a given RTL implementation actually upholds the μspec axioms of its microarchitectural specification. This enables PipeCheck's μspec microarchitectural ordering specifications to be validated against RTL implementations for soundness (in the case where the RTL exists prior to creating the μspec model), while also enabling verification of processor RTL against a μspec model's specified orderings (for the case where the μspec model is created prior to RTL).

Returning to the example of Multi-V-scale, Figure 3.3c's Verilog makes it appear as though instructions do indeed move to WB in the order in which they entered DX (thus satisfying the `WB_FIFO` axiom), but it is challenging to tell from inspection whether this is *always* the case. For instance, what if an interrupt occurs between the two instructions? The situation is exacerbated as design and axiom complexity increases, necessitating the creation of an automated tool like RTLCheck to automatically generate assertions that can be formally verified to check whether or not RTL maintains the required orderings.

3.3 RTLCheck Overview

Figure 3.4 shows the high-level flow of RTLCheck. Three of the primary inputs to RTLCheck are the RTL design to be verified, a μspec model of the microarchitecture,

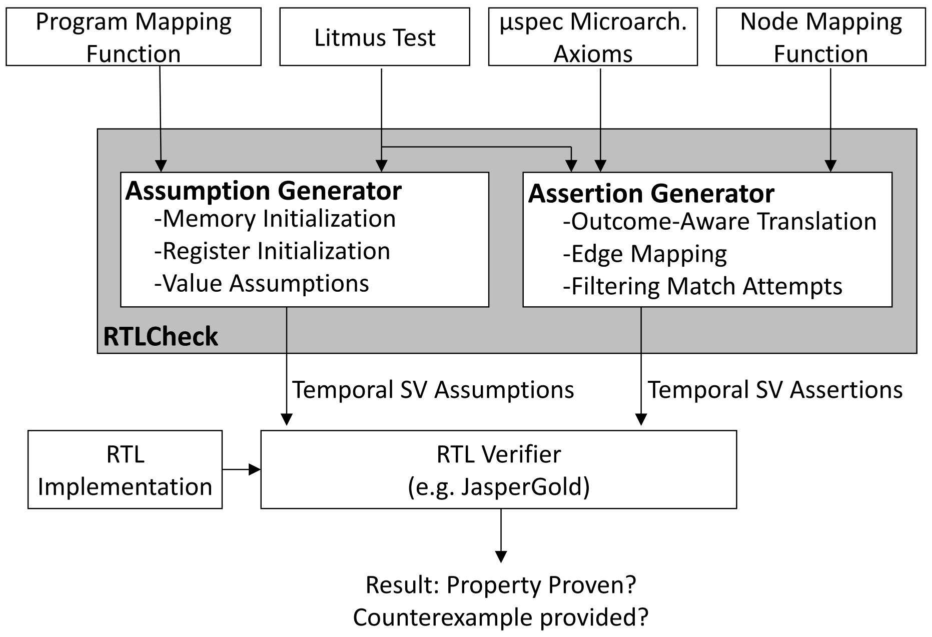


Figure 3.4: Overall flow diagram of RTLCheck.

and a suite of litmus tests to verify the design against. The other inputs to RTLCheck are the *program and node mapping functions* (described in Sections 3.6.1 and 3.6.3 respectively). The program mapping function translates litmus tests to initial/final state assumptions on the RTL being verified. These assumptions restrict the executions examined by the SVA verifier to those of the litmus test being verified. The node mapping function translates individual μhb nodes to RTL expressions describing the events modelled by those nodes. When translating a μspec axiom to an SVA assertion, the node mapping function enables RTLCheck to translate the μhb nodes and edges in that axiom to their RTL equivalents.

RTLCheck itself has two main components. The **Assumption Generator** (Section 3.6.1) generates SVA assumptions constraining the executions examined by a verifier to those of the litmus test being verified. It makes use of the program mapping function to do so. The **Assertion Generator** (Section 3.5.4) generates SVA assertions that check the individual axioms specified in the μspec model for the specific litmus test. It uses the node mapping function in its translation. When the Assertion Generator conducts its translation, it takes into account the differences between μspec and SVA semantics as well as the assumption over-approximation (Section 3.5) used by SVA verifiers.

The SVA assertions and assumptions that RTLCheck generates are passed to an SVA verifier (the RTLCheck flow uses Cadence JasperGold [Cad15b]), along with the RTL implementation. JasperGold then attempts to prove the assertions for the RTL implementation, subject to the assumptions. (Details of the JasperGold configurations that I use are provided in Section 3.9.2.) For each assertion, JasperGold may prove the assertion correct for all possible traces for that litmus test outcome (unbounded proof), for all traces of up to a certain number of cycles (bounded proof), or it may find a counterexample (execution trace that does not satisfy the property).

3.4 SystemVerilog Assertions (SVA) Background

RTLCheck automatically translates μspec axioms for a given litmus test to SystemVerilog Assertions (SVA) for that litmus test. This section provides necessary background on SVA, an industry standard for the formal verification of RTL [CDH+15, EF18]. The information in this section is necessary for understanding both RTLCheck's operation (Section 3.6) and the SVA over-approximation (Section 3.5) which RTLCheck must work around during its translation.

SVA is based on Linear Temporal Logic (LTL) (Section 3.4.1), but includes support for *regular expressions* (Section 3.4.2) and *suffix implication* (Section 3.4.3), which increase its expressive power and make it capable of specifying useful properties which LTL cannot express. This section draws heavily on the existing literature for LTL and SVA [BK08, EF18, CDH+15].

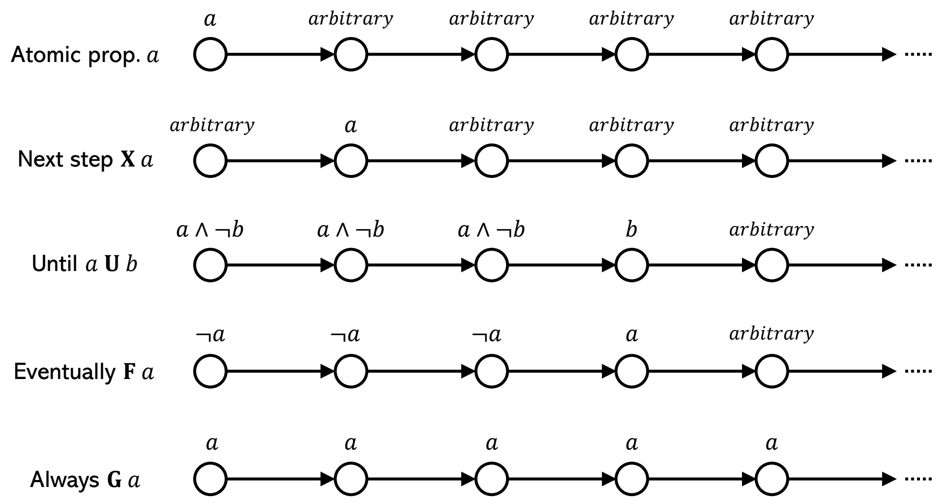


Figure 3.5: An illustration of the semantics of linear temporal logic properties, adapted from Baier and Katoen [BK08] with a few syntactic changes for consistency with the notation used in this dissertation.

3.4.1 Linear Temporal Logic (LTL)

LTL Syntax and Semantics

Temporal logic extends propositional logic with capabilities to refer to the execution of a system over time. Linear Temporal Logic (LTL) was created by Amir Pnueli in 1977 [Pnu77], and enables the specification of linear-time properties. Informally speaking, linear-time properties are properties that describe what the executions of a system should look like.

An abstract grammar for any LTL property φ is given below in Equation 3.1:

$$\varphi := \mathit{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathbf{X}\,\varphi \mid \varphi_1\,\mathbf{U}\,\varphi_2 \tag{3.1}$$

where a is an atomic proposition (i.e. a Boolean variable).

LTL properties are evaluated over traces of transition systems (Section 2.2.1), where the traces are of infinite length. Intuitively, the temporal operators like $\mathbf{X}$ and $\mathbf{U}$ allow the user to describe the behaviour of states that occur later in the trace than the current state.

Figure 3.5 illustrates the semantics of different types of LTL properties. An LTL property a where a is an atomic proposition is true if and only if the proposition a holds in the initial state of the trace. Logical connectives like $\neg$ and $\wedge$ function as they do in propositional logic. In other words, if p is an LTL property, then $\neg p$ is true over a given trace if and only if p is false over that trace. Similarly, if p and q are LTL properties, then $p \wedge q$ is true over a given trace if and only if p is true over that trace and q is also true over that trace.

As Figure 3.5 shows, an LTL property $\mathbf{X}\, a$ ("next") is true over a given trace t if and only if the LTL property a holds over the trace consisting of all of t except its first state. Meanwhile, an LTL property $a\, \mathbf{U}\, b$ ("until") holds if and only if a is true for every state in the trace until b becomes true. b may become true at any time (including in the first state), but it *must* eventually become true. In other words, even if a is always true over the infinite trace, if b never becomes true then $a\, \mathbf{U}\, b$ evaluates to false.

The until operator enables the creation of two other shorthands, $\mathbf{F}$ ("eventually") and $\mathbf{G}$ ("always") that are frequently used when writing LTL properties. $\mathbf{F}$ is defined for a given LTL property p as $\mathbf{F}\, p = \mathit{true}\, \mathbf{U}\, p$. In other words, $\mathbf{F}\, p$ is true if and only if the LTL property p is true at either the current state or some future state in the trace. $\mathbf{G}$, meanwhile, is defined for a given LTL property p as $\mathbf{G}\, p = \neg \mathbf{F}\, (\neg p)$. In other words, $\mathbf{G}\, p$ is true if and only if p holds at every state in the trace.

Safety and Liveness

LTL specifies linear-time properties, of which there are two major classes: *safety properties* and *liveness properties.* As Alpern and Schneider [Alp85] state, safety properties intuitively specify that "something bad will never happen" with respect to the system's behaviour, while liveness properties intuitively specify that "something good will happen". An example of a safety property is $\mathbf{G}\, p$, where p is an atomic proposition.[6] In this case, p becoming false is "something bad", and the property specifies that this should never happen (i.e. p must always be true). An example of a liveness property is $\mathbf{F}\,p$, where p is an atomic proposition. In this case, p becoming true is "something good", and the property specifies that this must happen at some point in the trace.

Alpern and Schneider [Alp85] also provide formal definitions of safety and liveness properties. They state that an LTL property p is a safety property if for every counterexample σ to p (i.e. an infinite trace over which p does not hold), there exists a finite prefix[7] $\hat{\sigma}$ of σ which cannot be extended to an infinite trace that satisfies p. In other words, for every counterexample to a safety property p there is a finite point in the counterexample trace where p becomes false and no further extension of that prefix can make the property hold.

Meanwhile, an LTL property p is a liveness property if for every finite prefix of an infinite trace, there exists an extension of it to an infinite trace such that the infinite trace so created will satisfy p [Alp85]. In other words, a finite prefix of an infinite trace is never irredeemable (unlike for safety properties). Counterexamples to liveness properties can be restricted (without loss of generality) to "lasso-shaped" traces, which consist of a finite prefix followed by an infinitely-repeated finite path. For instance, such a counterexample for the property $\mathbf{F}\,p$ is a sequence of states $s_0 s_1 s_2$ where p is false, followed by a repeating state s_3 where p is false. The resultant trace $s_0 s_1 s_2 s_3 s_3 s_3 ...$ is an infinite trace where p never holds, thus violating the property.

[6]Neither safety nor liveness properties are restricted to atomic propositions; the use of atomic propositions here is purely for the simplicity of the examples.

[7]A finite prefix of an infinite trace is a sequence of the first n states of that trace for some finite n. The remainder of the trace (i.e., the portion from the end of the prefix onwards) is known as the *suffix*.

Model Checking LTL Properties

As Baier and Katoen [BK08] state, model checking for LTL properties is generally conducted using an automata-based approach. To verify a given LTL property p on a transition system TS, the verification procedure constructs an automata (state machine) capable of recognising $\neg p$ (henceforth referred to as $A_{\neg p}$). It then takes the product of the transition system and state machine ($TS \otimes A_{\neg p}$).[8] The procedure then checks whether there is any trace of this product where $A_{\neg p}$ accepts. If so, the property does not hold and the trace where $A_{\neg p}$ accepts constitutes a counterexample. Otherwise, the property p holds on TS.

The specific type of automaton used for verification of general LTL properties is a nondeterministic Büchi automaton (NBA) [Büc90]. NBAs have no terminal states, but have an *acceptance set* that constitutes a subset of the automaton's states. The NBA accepts an infinite trace t if t causes the NBA to visit a state in the acceptance set infinitely often (i.e., at every point in the trace, one of the future states is in the acceptance set). Baier and Katoen [BK08] provide further details on automata-based LTL model checking.

If conducting verification of LTL properties using symbolic model checking, then the above procedure requires the calculation of a nested fixed-point expression over the states of the system, and is often very difficult [KV99]. The calculation of the fixed-point expression corresponds to the search for a bad cycle in the system's state space. However, if one restricts oneself to the verification of LTL safety properties, a more computationally inexpensive approach can be used [KV99]. As stated above (in the "Safety and Liveness" section), every counterexample to a safety property has a finite "bad" prefix after which it can never be extended to a trace satisfying the property. Thus, instead of using an NBA to verify an LTL safety property p_{safety} on a

[8]The product $TS \otimes A$ of a transition system TS and automaton A consists of TS and A moving together in lockstep, with A transitioning according to the labels of the states that TS traverses [BK08].

transition system TS, one can construct a *finite* automaton A_{p_prefix} that recognises at least one bad prefix of every counterexample to p_{safety}. The verification procedure can then simply take the product $TS \otimes A_{p\_prefix}$, and check that there is no trace of this product that reaches an accepting state of A_{p_prefix}. This can be done through symbolic reachability analysis, which is notably simpler than the NBA case [KV99].

The lower difficulty of the verification of safety properties as opposed to properties that include liveness is the motivation for the SVA verifier over-approximation (Section 3.5), which is the major source of difficulty for RTLCheck's translation of μspec to SVA.

3.4.2 Regular Expressions

While LTL is quite an expressive logic, there are certain types of properties desirable for RTL verification that LTL cannot express on its own. For instance, it is impossible to state the property "p holds at every even position[9]" in LTL [Wol81]. To fill this gap, the SVA specification includes support for regular expressions and suffix implication. In formal methods terminology, the combination of LTL with regular expressions and suffix implication has the expressive power of ω-regular languages, while LTL alone only has the expressive power of star-free ω-regular languages [EF18] (a strict subset of the ω-regular languages).[10]

Regular expressions are expressions that describe regular languages, which are themselves the languages accepted by finite automata [Sip06]. Regular expressions are often used to describe patterns of strings in computer science. They include the capability to express repetition (*), alternation ($\mid$), and concatenation($\cdot$). If R is a regular expression, then R^* matches 0 or more repetitions of R. If R_1 and R_2 are regular expressions, then $R_1 \mid R_2$ matches either R_1 or R_2. Finally, if R_1 and R_2 are regular expressions, then $R_1 \cdot R_2$ matches a string consisting of a string that matches

[9]Here, a "position" refers to a state in a path or trace.
[10]In this section, a "language" is a set of strings [Sip06].

R_1 followed by a string that matches R_2. For example, the regular expression `(a|b)*·c` will match strings including `aaac`, `bbc`, and `ababbc`. Sipser [Sip06] provides further details on regular expressions.

3.4.3 Suffix Implication

Suffix implication is similar to regular implication, but with the difference that the consequent is only evaluated for the portion of the infinite trace that starts at the point the antecedent matches. In a suffix implication $r \mapsto q$, r is a regular expression and q is a property of LTL extended with regular expressions and suffix implication. The implication is true over an infinite trace t if for every prefix t_{pre} of t that matches r, the suffix of t that starts at the point where t_{pre} ends matches q. For example, the property $(\neg a \cdot \neg a \cdot \neg a) \mapsto \mathbf{X}\, a$ is true when evaluated on the fourth trace in Figure 3.5. This is because for the prefix of the trace that consists of 3 successive states where where a is false (i.e. the first 3 states of the trace), a is true in the next (fourth) state.

As in regular implication, if the antecedent does not match, then the property evaluates to true. So for instance, the property $(\neg a) \cdot (\neg a)^* \mapsto b$ evaluates to true on the last trace in Figure 3.5 because the implication's antecedent never matches any prefix of the trace (since a is true in all states of the trace).

The combination of LTL, regular expressions, and suffix implication can state the property "p holds at every even position" as follows [BBDL98]:

$$(true \cdot (true \cdot true)^*) \mapsto p$$

SVA is based on LTL, and includes support for regular expressions and suffix implication. The next section provides an overview of SVA assertions and assumptions, with examples.

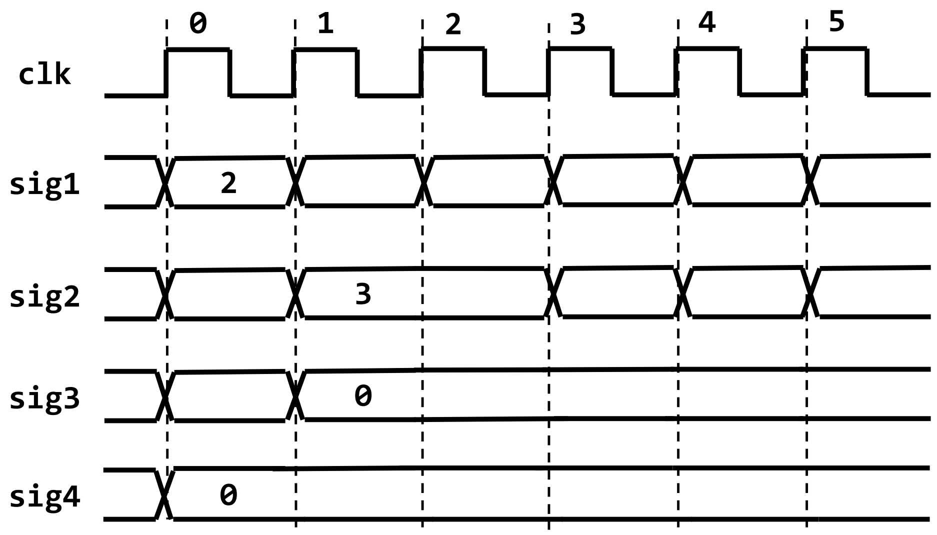


Figure 3.6: An example trace of four RTL signals with their clock. Section 3.4.4 evaluates an example assertion and assumption on this trace.

3.4.4 SVA Assertions and Assumptions

In SVA, atomic propositions are expressions over the signals in RTL that use common arithmetic and logical operators. So for instance, if `sig1` and `sig2` are RTL signals, then (`sig1 == 1 && sig2 == 2`) is an atomic proposition. A state consists of a snapshot of the values of all RTL signals in a single clock cycle. The LTL operators **X**, **U**, **G**, and **F** all have analogues in SVA: `s_nexttime`, `s_until`, `always`, and `s_eventually`. SVA also includes some syntactic sugar for a few other LTL operator combinations [IEE13].

SVA includes capabilities for specifying regular expressions, called *sequences.* A sequence may be an atomic proposition, or it may be created from other sequences. A sequence matches a trace if some prefix of the trace matches the pattern described by the sequence. The operators `and`, `or`, and `not` can take sequences as their operands to create new sequences. For instance, `a and b` is a sequence which matches a trace if both `a` and `b` match it. Sequences can be concatenated together with delays between them. `##<x>` represents a delay of `x` cycles, so for example, `a ##2 b` matches a trace if `a` matches the trace, and 2 cycles after the end of a match of `a`, the remaining suffix of the trace matches b. Delays can have ranges; ##[<x>:<y>] represents a delay of between x and y cycles. y may be $, in which case the maximum number of delay cycles is unbounded. Finally, sequences can be repeated some number of times. seq [*<x>:<y>] represents between x and y consecutive repetitions of the sequence seq. Again, y may be $, in which case the maximum number of repetitions is unbounded. Suffix implication in SVA makes use of SVA's support for regular expressions. In SVA, suffix implication is specified using |->, and its semantics are exactly as explained in Section 3.4.3 above.

Clocks provide a natural notion of time in RTL. Indeed, in SVA, properties are given a clock, and move forward one step in time on the edge of the clock. In other words, the LTL notion of moving from one state to the next state in a trace corresponds to signals taking their new values at the next clock edge in SVA.

To put it all together, consider the following example SVA property:

```
assert property (@(posedge clk) ((sig1 == 2) ##1 (sig2 == 3) [*1:$]) |->
    always (sig3 == 0));
```

The property matches traces where in the first cycle, sig1 has a value of 2, followed by 1 or more cycles where sig2 has a value of 3, and at the end of each of these prefixes, sig3 has a value of 0 for the remainder of the trace. This property will evaluate to true on the RTL execution shown in Figure 3.6 (assuming the signal values never change after the cycles shown). In Figure 3.6, sig1 has a value of 2 in cycle 0 as required, and in cycles 1 and 2, sig2 has a value of 3. Therefore, 2 prefixes match the antecedent of the implication: cycles 0-1, and cycles 0-2. sig3 is 0 from cycle 1 onwards, and so always (sig3 == 0) evaluates to true at cycle 1 (required by the first prefix) and cycle 2 (required by the second prefix). Since for all prefixes matching the antecedent, the consequent is satisfied by the corresponding suffix of the trace, the overall property evaluates to true.

By default, SVA will evaluate the property on traces that start at every cycle of the execution (rather than just the first cycle). This can be thought of as putting an `always` around the overall property. To evaluate the property only once at the start of the RTL execution, one may put the assertion in an `initial` block in Verilog [CDH$^+$15], or use implication appropriately as RTLCheck does. (See Sections 3.6.1 and 3.6.5 for details.)

Users may often wish to restrict the set of RTL executions they are verifying in some way. For instance, RTLCheck's assertions are litmus test-specific, so it needs to restrict the verifier to those executions where instruction memory is initialised with the test program. (RTLCheck also requires other restrictions to the considered executions; see Section 3.6.1 for details.) SVA provides users with the ability to do this through *assumptions*. SVA assumptions have the same syntax as SVA assertions, but use the `assume` keyword rather than `assert`. If provided with an assumption, SVA verifiers ignore RTL executions that caused the assumption to become false. For instance, the assumption

```
assume property (@(posedge clk) (sig4 == 1));
```

will restrict an SVA verifier to only those executions where `sig4` always has a value of 1. This would cause the verifier to ignore Figure 3.6's execution.

The handling of SVA assumptions by SVA verifiers uses an over-approximation which makes translation from μspec to SVA difficult, as discussed in the next section.

3.5 Handling the SVA Verifier Assumption Over-Approximation

When verifying SVA safety properties (Section 3.4.1) under a set of SVA assumptions, SVA verifiers like JasperGold use an over-approximation of the assumptions to improve performance. In a nutshell, SVA verifiers attempt to prove safety properties under *weaker* assumptions than those which the user provides. Thus, the verifier effectively attempts to prove a stronger property than that which the user specified. The over-approximation can provide improved RTL verification performance for safety properties. However, it introduces false positives when RTL satisfies the user-specified property but not the over-approximation.

A straightforward translation of μspec axioms for a litmus test to SVA assertions and assumptions suffers from false positives because of this over-approximation. Under such a translation scheme, when the SVA verifier returns a counterexample to an RTLCheck-generated SVA assertion, the RTL may indeed be buggy, or it may be a false positive. This makes it very hard (if not impossible) for an automated tool like RTLCheck to determine the correctness of the RTL using such a translation scheme. RTLCheck therefore takes care to translate μspec axioms for a litmus test to SVA assertions in a way that works around this over-approximation, eliminating the false positives.

This section covers the over-approximation in detail, including the reasoning behind it (Section 3.5.1), a discussion of its semantics (Section 3.5.2), how it can cause problems when conducting MCM verification (Section 3.5.3), and RTLCheck's solution that works around the over-approximation (Section 3.5.4).

3.5.1 Reasoning Behind the Over-Approximation

Consider the following SVA fragment (example taken from Cerny et al. [CDH+15]):

```
assume property (a);
assert property (b);
```

This is equivalent to:

```
assert property ((always a) implies (always b));
```

or to use LTL notation, $\mathbf{G}\, a \rightarrow \mathbf{G}\, b$. This is in turn equivalent to $\mathbf{F}\, \neg a \vee \mathbf{G}\, b$. This property is a liveness property (Section 3.4.1). To see why, note that any finite trace prefix that seemingly violates the property (by having b become false at some cycle in the prefix) can always be extended to a trace satisfying the property simply by having a be false in a future cycle. Having a be false in a future cycle makes $\mathbf{F}\, \neg a$ true, which is enough to satify the property.

Thus, to prove an assertion b subject to an assumption a, one must prove $\mathbf{F}\, \neg a \vee \mathbf{G}\, b$, which is a liveness property. $\mathbf{F}\, \neg a \vee \mathbf{G}\, b$ will be a liveness property even if b is a safety property. As such, the existence of any assumption a for a safety property b turns the safety property b into a liveness property. This prevents SVA verifiers from verifying safety properties using the optimized verification procedure that looks for bad prefixes (Section 3.4.1), and can significantly increase verification runtime. Rather than verify all properties as liveness properties, SVA verifiers like JasperGold choose to compromise by weakening assumptions enough to remove their liveness component. This reverts the proof of a safety property b under the assumption a to a proof of a safety property [CDH+15, Cad15a], which can be verified using the optimized verification procedure explained in Section 3.4.1.

Next, I discuss the exact nature of the over-approximation.

3.5.2 The Assumption Over-Approximation

Section 3.4.1's optimized verification procedure for a safety property b searches for a finite bad prefix violating b. If b is subject to an assumption a, the assumption over-approximation only requires a to hold for the cycles in the bad prefix (as opposed to all cycles in the infinite trace) for the bad prefix to be a counterexample. Thus, any finite bad prefix violating b can no longer be extended to a trace satisfying the overall property by having a become false after the bad prefix ends, because the assumption over-approximation does not enforce constraints on a after the end of the bad prefix.

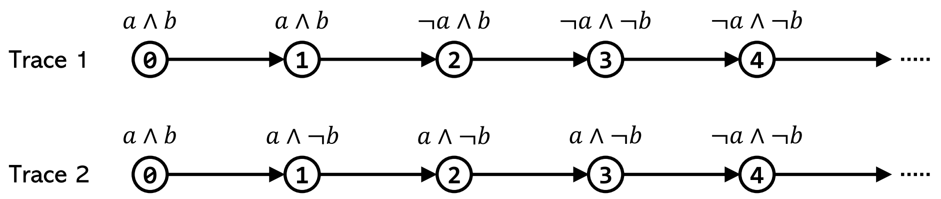


Figure 3.7: Two infinite traces over which an assertion b is evaluated subject to an assumption a. Each cycle is annotated with its cycle number as well as whether a and b hold in that cycle. b holds over Trace 1 regardless of whether the assumption over-approximation is used. However, on Trace 2, b only holds if the over-approximation is not used. If the over-approximation is used, an SVA verifier will return a prefix of Trace 2 as a (false positive) counterexample to b.

As a result, all counterexamples to the verification of a safety property b subject to an assumption a are now finite. This makes the verification of b subject to a a safety property, which can be verified using Section 3.4.1's optimized procedure.

The assumption over-approximation leads to JasperGold verifying a property that is strictly stronger than required [CDH+15]. Counterexamples to the exact property require b to be false for some cycle in the trace and require a to be true for the entire (infinite) trace. Meanwhile, counterexamples to the property when using the over-approximation only require b to be false for some cycle in the trace and for a to be true for all the cycles up to and including the cycle where b becomes false.

Consider verifying a safety property b subject to an assumption a on the traces in Figure 3.7. Each cycle (state) in the trace is labelled with its cycle number, as well as whether a and b hold starting at those cycles. The top trace satisfies the exact property $\mathbf{F}\,\neg a \vee \mathbf{G}\,b$, because a becomes false at cycle 2. It also satisfies the property if the over-approximation is used. This is because a becomes false for the first time before b becomes false for the first time, so there is no bad prefix containing a cycle where b is false but where a is true for the entirety of the prefix. (The result would be the same if a and b both became false for the first time at the same cycle.)

The bottom trace also satisfies the exact property $\mathbf{F}\,\neg a \vee \mathbf{G}\,b$, because a becomes false at cycle 4. However, it does *not* satisfy the property if the over-approximation is used, in which case it becomes a false positive. This is because b becomes false for the first time at cycle 2, before a becomes false for the first time at cycle 4. Thus, cycles 0-2 constitute a bad prefix violating b where a is true for every cycle in the prefix, and this qualifies as a counterexample when using the over-approximation.

Theoretically, a workaround to the over-approximation would be to not use assumptions at all, and instead to include the required assumptions inside every assertion. For example, instead of writing:

```
a_1: assume property (@(posedge clk) ...);
a_2: assume property (@(posedge clk) ...);
...
a_n: assume property (@(posedge clk) ...);


assert property (@(posedge clk) <assertion body>);
```

one would write:

```
assert property (@(posedge clk)
  (always a_1 and always a_2 and ... and always a_n) implies <assertion body>);
```

However, if RTLCheck generates properties in this manner, they prove to be practically infeasible for the JasperGold verifier. When JasperGold conducts verification, it first compiles the SVA assertions provided to it and then begins their verification. If generating assertions which internally contain their assumptions (as shown above), it takes JasperGold over 10 hours to merely *compile* the generated assertions for the Multi-V-scale processor (RTLCheck's case study, explained in Section 3.7) for the large litmus test `amd3`. In other words, it takes over 10 hours before verification of the RTL for `amd3` even begins. On the other hand, if generating assertions in the traditional manner (with assertions and assumptions separate), compilation finishes in seconds or minutes for the litmus tests I evaluated, and 10 hours of runtime is enough to prove most of the properties RTLCheck generates for Multi-V-scale. Section 3.9 provides further details on RTLCheck's runtimes.

RTLCheck's solution to work around the over-approximation (Section 3.5.4) is to generate SVA assertions that only require the weaker assumptions permitted by the over-approximation, rather than the more powerful assumptions that would be available if the over-approximation were not used. The next section explains how the over-approximation can affect MCM verification of RTL.

3.5.3 The Over-Approximation in MCM Verification

RTLCheck's translation of μspec axioms to equivalent SVA assertions and assumptions for a litmus test is a two-step process:

1. Translate the litmus test to SVA assumptions that restrict the SVA verifier to RTL executions of the litmus test (Section 3.6.1), and

2. Translate the μspec axioms to equivalent SVA assertions for the litmus test (Section 3.5.4).

When verifying a litmus test on a μspec model with PipeCheck [LPM14, LSMB16], the litmus test serves as a set of additional constraints, restricting PipeCheck to only look at executions of that litmus test with the specified outcome (e.g. `r1=1,r2=0` for `mp`). When verifying μspec axioms at RTL, SVA assumptions perform the same function. However, RTLCheck's translated assumptions are *weakened* by the assumption over-approximation such that that they no longer enforce all the constraints that the litmus test constraints do for microarchitectural verification. Specifically, it becomes infeasible to restrict the SVA verifier to a specific litmus test outcome (e.g. `r1=1, r2=0` for `mp`). Thus, if RTLCheck relies on all the constraints of the litmus test when translating μspec axioms to SVA assertions, the generated assertions will fail (false

```
DefineMacro "BeforeAllWrites":
  DataFromInitialStateAtPA i /\
  forall microop "w", (
    (IsAnyWrite w /\ SameAddress w i /\ ~SameMicroop i w) =>
    AddEdge ((i, Writeback), (w, Writeback), "fr", "red")).

DefineMacro "NoInterveningWrite":
  exists microop "w", (
    IsAnyWrite w /\ SameAddress w i /\ SameData w i /\
    EdgeExists ((w, Writeback), (i, Writeback)) /\
    ~(exists microop "w'",
      IsAnyWrite w' /\ SameAddress i w' /\ ~SameMicroop w w' /\
      EdgesExist [((w , Writeback), (w', Writeback), "");
                  ((w', Writeback), (i, Writeback), "")])).

DefineMacro "BeforeOrAfterEveryWrite":
  forall microop "w", (
    (IsAnyWrite w /\ SameAddress w i) =>
    (AddEdge ((w, DecodeExecute), (i, DecodeExecute)) \/
     AddEdge ((i, DecodeExecute), (w, DecodeExecute)))).

Axiom "Read_Values":
forall microops "i",
OnCore c i => IsAnyRead i => (
      ExpandMacro BeforeAllWrites
      \/
      (
        ExpandMacro NoInterveningWrite
        /\
        ExpandMacro BeforeOrAfterEveryWrite
      )
).
```

Figure 3.8: μspec axiom enforcing orderings and value requirements for loads on the Multi-V-scale processor. Predicates relevant to load values are highlighted in red. The edge referred to in Section 3.6.3 is highlighted in blue.

positives) because the assumptions under which they are verified have been weakened. This section illustrates how such false positives can arise when translating μspec axioms to SVA assertions and assumptions for a given litmus test.

Figure 3.8 shows a μspec axiom (and related macros) from the Multi-V-scale microarchitecture definition. This axiom splits the checking of load values into two categories: those which read from some write in the execution, and those which read from the initial value of the address in memory. The axiom is litmus-test-independent: it applies equally to any program that runs on the microarchitecture being modeled.

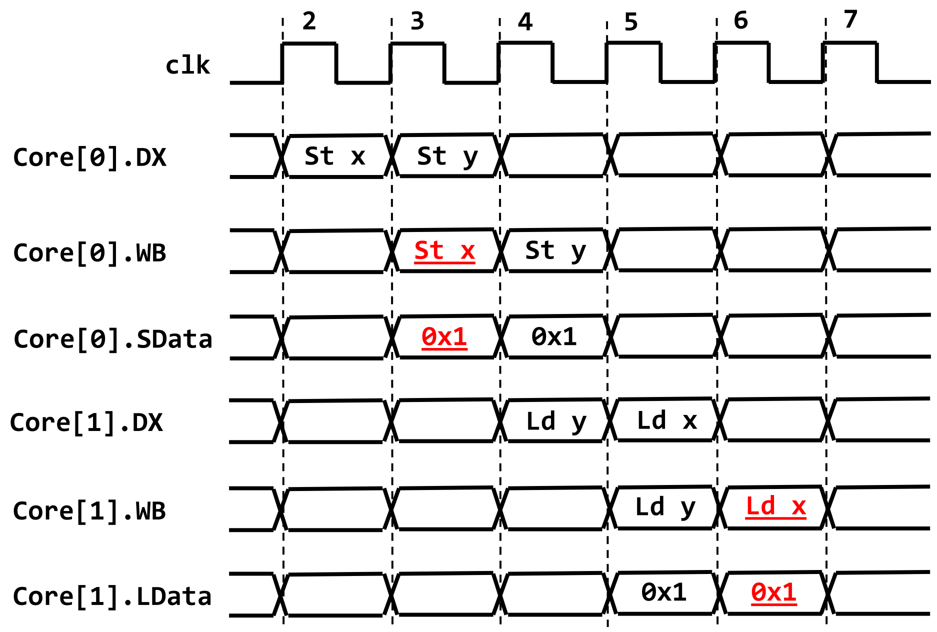


Figure 3.9: Example execution trace of `mp` on Multi-V-scale where `Ld y` returns 1 and `Ld x` returns 1. Signals relevant to the events `St x @WB` and `Ld x @WB` are underlined and in red.

However, if verifying a single litmus test, PipeCheck uses the information about the litmus test outcome of interest to prune out all logical branches of the axiom which do not result in that outcome. Consider the application of this axiom to the load of `x` from `mp`, which returns 0 in the outcome under test. In μspec, the `SameData w i` predicate evaluates to true if instructions `w` and `i` have the same data in the litmus test outcome, while `DataFromInitialStateAtPA i` returns true if `i` reads the initial value of a memory location. For `mp`, PipeCheck evaluates all data-related predicates (highlighted in red in Figure 3.8) according to the outcome specified by the litmus test. For the load of `x`, PipeCheck evaluates the `SameData w i` predicate in the `NoInterveningWrite` part of the axiom to false (as there is no write which stores 0 to `x` in `mp`). This causes the `NoInterveningWrite /\ BeforeOrAfterEveryWrite` portion of the `Read_Values` axiom to evaluate to false. Thus, the body of the `Read_Values` axiom is reduced to `BeforeAllWrites`, and PipeCheck knows that it must add an edge indicating that $\texttt{Ld x @WB} \xrightarrow{hb} \texttt{St x @WB}$[11] (shown as one of the red edges in Figure 3.3a).

If translating this simplified axiom (and the relevant part of the litmus test outcome) to an SVA assertion `p2` and assumption `p1`, the result is:

```
p1: assume property (@(posedge clk) <Ld x returns 0>);
p2: assert property (@(posedge clk) <Ld x@WB happens before St x@WB>);
```

The specific SVA constructs used for the assumption and assertion are covered in Section 3.6.

Now consider evaluating `p2` subject to the assumption `p1` on a correct implementation of Multi-V-scale. Figure 3.9 shows one possible trace of `mp` on Multi-V-scale, specifically one where both loads return 1 (i.e. `r1=1,r2=1`). (No events of interest happen in cycles 0-1 of the trace, so they are not shown for brevity.) This is clearly correct behaviour on the part of Multi-V-scale, as instructions are performed in-order and atomically, and the outcome `r1=1,r2=1` is allowed under SC. Without the assumption over-approximation, an SVA verifier would correctly ignore this trace because the load of `x` returns 1 in it, thus violating the assumption `p1`. However, with the assumption over-approximation, cycles 0-3 of Figure 3.9's trace will be returned to the user as a counterexample to `p2`, which constitutes a false positive. The reason for this is as follows. `p2` is a safety property (Section 3.6.3 provides details on the exact SVA constructs used to map the edge). Cycle 3 violates `p2`, because the store of `x` has happened before the load of `x`. Thus, cycles 0-3 constitute a bad prefix to the safety property `p2`. Under the assumption over-approximation, the assumption `p1` only needs to hold for the cycles that are part of the bad prefix. The load of `x` does not happen in cycles 0-3, so there is no need to enforce its value as being 0 in those cycles. Thus, the assumption `p1` is satisfied for cycles 0-3, which is enough

[11]In this chapter, $a \xrightarrow{hb} b$ denotes a μhb edge from a to b.

for these cycles to constitute a counterexample to `p2` under the over-approximation. The assumption is indeed violated in cycle 6, so Figure 3.9's trace should not be a counterexample. Indeed, on a correct implementation of Multi-V-scale, it is impossible to extend cycles 0-3 to an infinite trace[12] where the load of `x` returns 0. However, the over-approximation ignores this fact and returns cycles 0-3 of Figure 3.9's trace as a counterexample.

To overcome the challenge introduced by the assumption over-approximation, RTLCheck's translation procedure must translate μspec axioms for a litmus test in a way that accounts for *all* outcomes of the litmus test, not just its outcome of interest. The next section describes how RTLCheck does so.

3.5.4 Solution: Outcome-Aware Assertion Generation

Due to the assumption over-approximation, the assertion generated for the axiom in Figure 3.8 for `mp`'s load of `x` must account for both the case where the load of `x` returns 1 *and* the case where it returns 0. (The load of `x` cannot return any other values in an execution of `mp`.)

If the load of `x` returns the initial value of 0, this corresponds to the `BeforeAllWrites` portion of the axiom. The SVA property for this case must check that `Ld x @WB` $\xrightarrow{hb}$ `St x @WB` and that the load returns 0. Similarly, if the load returns 1, this corresponds to the part of the axiom comprising `NoInterveningWrite` and `BeforeOrAfterEveryWrite`. The equivalent SVA property here must check that `St x @WB` $\xrightarrow{hb}$ `Ld x @WB` and that the load returns 1.

To accomplish this outcome-aware translation of μspec axioms, RTLCheck does not evaluate data-related predicates like `SameData` or `DataFromInitialStateAtPA` to true or false based on the litmus test it is generating properties for. Instead, for

[12]Assume that the transition system representing the RTL enters a halt state once the litmus test has finished executing, and that the only next state for the halt state is itself. Thus, an infinite trace of the RTL's transition system constitutes an execution of the litmus test followed by an infinite repetition of the halt state.

each such predicate, if any edge requires the predicate to be true for the edge to hold, RTLCheck generates a *load value constraint* for the edge based on the value of the data-related predicate. Load value constraints encode the constraints on the data values of loads involved in a given edge, and must be taken into account by the node mapping function when translating individual μhb edges (Section 3.6.3).

For `mp`'s load of `x`, the μhb edge in Figure 3.8's `BeforeAllWrites` macro requires `DataFromInitialStateAtPA i` to be true, which in turn requires that the load returns the initial value 0. This requirement on the load's value is stipulated as a load value constraint when mapping the edge from `BeforeAllWrites`. Similarly, the μhb edges in Figure 3.8's `NoInterveningWrite` and `BeforeOrAfterEveryWrite` macros require `SameData w i` to be true. In the case of `mp`'s load of `x`, this predicate enforces that the load returns 1 (the same data as the write `i1` in `mp`), and this requirement is also encoded as a load value constraint when mapping the edge. The mapped edges of the two cases are combined with an SVA `or` (translated from the μspec `\/`), allowing the translated property to cater to both the executions where the load of `x` returns 0 and the executions where it returns 1, as necessary under the assumption over-approximation. (Section 3.6.2 provides further details on RTLCheck's overall axiom translation procedure.)

Special handling is also required for the μspec predicate `DataFromFinalStateAtPA`. The predicate `DataFromFinalStateAtPA i` returns true if `i` stores a value equivalent to the final value of its address in the litmus test. Accounting for this predicate at RTL requires ensuring that a given write is the last write to a particular address in an execution. However, just as in the case of ensuring a specific litmus test outcome, the assumption over-approximation makes it infeasible to ensure that a particular write happens last in an RTL execution. Thus, when translating this predicate, assertion generation always (conservatively) evaluates this predicate to false. Doing so ensures that the generated properties check all possible orderings of writes (i.e., a superset

```
assume property (@(posedge clk) first |-> mem[21] == {32'd0});
assume property (@(posedge clk) first |->
  mem[1] == {7'b0,5'd2,5'd1,3'd2,5'b0,`RV32_STORE});
assume property (@(posedge clk)
  (((core[1].PC_WB == 32'd24 && ~(core[1].stall_WB)) |->
    (core[1].PC_WB == 32'd24 && ~(core[1].stall_WB) &&
    core[1].load_data_WB == 32'd1))
  and
  ((core[1].PC_WB == 32'd28 && ~(core[1].stall_WB)) |->
    (core[1].PC_WB == 32'd28 && ~(core[1].stall_WB) &&
    core[1].load_data_WB == 32'd0)))
);
assume property (@(posedge clk)
  (((core[0].halted == 1'b1 && ~(core[0].stall_WB)) &&
    (core[1].halted == 1'b1 && ~(core[1].stall_WB)) &&
    (core[2].halted == 1'b1 && ~(core[2].stall_WB)) &&
    (core[3].halted == 1'b1 && ~(core[3].stall_WB))) |-> (1)));
```

Figure 3.10: A subset of the SV assumptions RTLCheck generates for `mp`. Some signal structure is omitted for brevity.

of the executions that PipeCheck would examine), including the write ordering the litmus test focuses on.

3.6 RTLCheck Operation

This section describes the operation of RTLCheck (Figure 3.4) in detail. Section 3.6.1 describes the process of assumption generation, and Section 3.6.2 describes the overall procedure for assertion generation. Sections 3.6.3 and 3.6.4 describe the process of mapping individual μhb edges and `NodeExists` predicates to SVA respectively. Finally, Section 3.6.5 describes how RTLCheck ensures that generated assertions are checked from the beginning of traces and not for some suffix of a trace.

3.6.1 Assumption Generation

RTLCheck's generated properties are litmus test-specific, so the executions examined by an RTL verifier for these properties need to be restricted to the executions of the litmus test in question. As depicted in Figure 3.4, the Assumption Generator performs this task using a *program mapping function* provided by the user. Program mapping functions link a litmus test's instructions, initial conditions, and final values of loads and memory to RTL expressions representing these constraints on the implementation to be verified. The parameters provided to a program mapping function are the litmus test instructions, context information such as the base instruction ID for each core, and the initial and final conditions of the litmus test.

The assumptions generated for a given litmus test must accomplish three tasks:

1. Initialize data and instruction memories to the litmus test's initial values and instructions respectively.

2. Initialize registers used by test instructions to appropriate address and data values.

3. Enforce that the values of loads and the final state of memory respect test requirements in generated RTL executions (to the extent allowed by the assumption over-approximation).

Figure 3.10 shows a subset of the assumptions that must be generated for the `mp` litmus test from Figure 3.2 for Multi-V-scale.

Memory Initialization: The first assumption in Figure 3.10 is an example of data memory initialization. It sets `x` to its initial value of 0 as required by `mp`. The `first` signal is auto-generated by the Assumption Generator, and is set up so that it is 1 in the first cycle after reset and 0 on every subsequent cycle. By using suffix implication triggered on the `first` signal being 1, the assumption only enforces that the value of the address in memory is equivalent to the initial condition of the litmus test at the beginning of the execution. This distinction is necessary as the verification needs to allow the address to change value when stores write to that address as the execution progresses. (The `first` signal is also used to filter match attempts as Section 3.6.5 describes.) The second assumption is an instruction initialization assumption. It enforces that core 0's first instruction is the store that is `i1` in Figure 3.2's `mp` code.

Register Initialization: The assembly encoding of litmus test instructions uses registers for addresses and data. Register initialization assumptions set these registers to the correct addresses and data values at the start of RTL execution. They are similar in structure to memory initialization assumptions.

Value Assumptions: Load value assumptions cannot be used to enforce an execution outcome due to the assumption over-approximation of SVA verifiers (see Section 3.5), but they can still be used to guide the verifier and reduce the number of executions it needs to consider. The third assumption in Figure 3.10 contains two such implications. Each one checks for the occurrence of one of the loads in `mp` and enforces that it returns the value in the test outcome (0 and 1 for `x` and `y` respectively) when it occurs. The last assumption in Figure 3.10 is a final value assumption. It contains an implication whose antecedent is the condition that all cores have halted their Fetch stages and all test instructions have completed their execution. The consequent of the implication stipulates any final values of memory locations that are required by the litmus test. Since `mp` does not enforce any such requirements, the consequent of the implication is merely a `1` (i.e. "true").

A pleasantly surprising side effect of assumption generation is that for certain tests, assumptions alone turn out to be sufficient in practice to verify the RTL. For most assumptions, JasperGold (the commercial RTL verifier used by RTLCheck) can find *covering traces*, which are traces where the assumption condition occurs and can be enforced. For instance, a covering trace for an assumption enforcing that the load of `y`

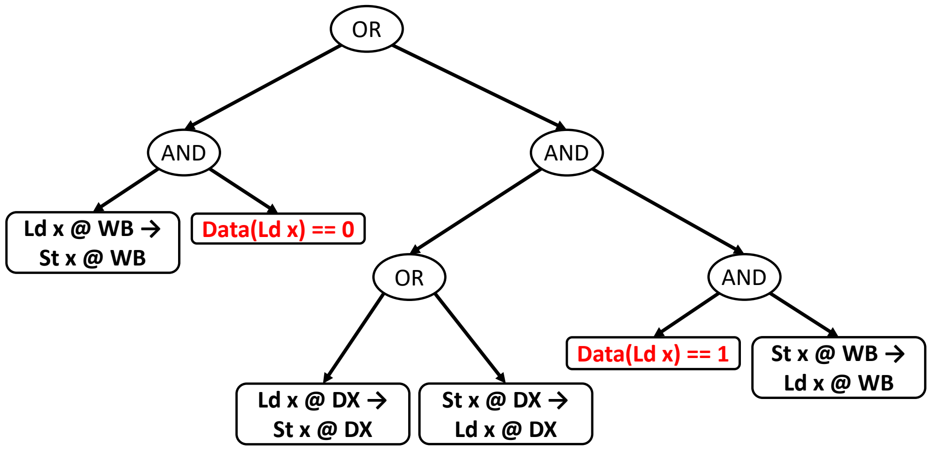


Figure 3.11: The constraint tree generated by RTLCheck's parsing and simplification of Figure 3.8's `Read_Values` axiom for the load of `x` in `mp`. Load value constraints are highlighted in red.

returns 1 would be a partial execution where the load of `y` returns 1 in the last cycle. A covering trace for a final value assumption in particular would by definition contain the execution of *all* instructions in the test. The covering trace must also obey any constraints on instruction execution stipulated by other assumptions, including load value assumptions. As such, a covering trace for `mp`'s final value assumption is an execution where the load of `y` returns 1 and the load of `x` returns 0. A search for such a trace is equivalent to finding an execution where the entire forbidden outcome of `mp` occurs. If JasperGold can prove that a covering trace for an assumption *does not exist*, it will label the assumption as unreachable. An unreachable final value assumption means that there are no executions satisfying the test outcome. This result verifies the RTL for that litmus test without checking the generated assertions. Thus, a final value assumption forces JasperGold to try and find a covering trace of the litmus test outcome, possibly leading to quicker verification. As such, final value assumptions are beneficial even when the test does not specify final values of memory, but I expect this benefit to be largest in relatively small designs. Section 3.9.2 quantifies my results.

3.6.2 Overall μspec Axiom Translation Procedure

RTLCheck's translation of μspec axioms to SVA assertions begins by parsing and simplifying the μspec axioms provided to it as input. This parsing and simplification is based on that of PipeCheck (explained in Section 2.4), but with two notable differences. Specifically, as Section 3.5.4 explains, RTLCheck does not evaluate the data-related predicates `SameData` and `DataFromInitialStateAtPA` to true or false for a given litmus test. Instead, RTLCheck determines what value a load must have in that litmus test in order for these predicates to be true, and evaluates the predicate to a load value constraint which encodes this information. Additionally, RTLCheck conservatively evaluates the `DataFromFinalStateAtPA` predicate in any μspec axiom to false.

RTLCheck evaluates all other μspec predicates to true or false according to the litmus test it is generating assertions for. The result of this parsing and simplification is a tree of constraints which must be satisfied by the RTL in order for it to maintain the μspec axiom for the litmus test. Nodes in this constraint tree may be `AND`, `OR`, and `NOT` (parsed from the μspec `/\`, `\/`, and `~` respectively), μhb edges or nodes stipulated by the μspec axiom, or load value constraints generated from the evaluation of data-related predicates.

For example, Figure 3.11 shows the constraint tree generated after the parsing and simplification of Figure 3.8's `Read_Values` axiom for the load of `x` in `mp`. This constraint tree has two branches, each corresponding to a different value for the load of `x` in `mp`. The branches contain load value constraints corresponding to those outcomes. The left branch of the top-level `OR` corresponds to the `BeforeAllWrites` case of the `Read_Values` axiom. For the `BeforeAllWrites` case to be true, the edge $\texttt{Ld x@WB} \xrightarrow{hb} \texttt{St x@WB}$ must exist. The predicate `DataFromInitialStateAtPA i` must also evaluate to true where `i` is the load of `x`, which means that the load of `x` must return 0. RTLCheck duly encodes this information as a load value constraint for that branch.

Meanwhile, the right branch of the top-level `OR` corresponds to the `NoInterveningWrite /\ BeforeOrAfterEveryWrite` portion of the `Read_Values` axiom. This branch of the axiom requires the edge `St x@WB`$\xrightarrow{hb}$`Ld x@WB` to exist, as well as one of the `Ld x@DX`$\xrightarrow{hb}$`St x@DX` and `St x@DX`$\xrightarrow{hb}$`Ld x@DX` edges. RTLCheck therefore includes these edges in the constraint tree's right branch, as Figure 3.11 shows. This branch of the axiom also requires `SameData w i` to be true, where `w` is a write to the same address[13] as `i` and `i` is the load of `x`. For this `SameData` predicate to be true, the load of `x` must have the same value as the store to `x` in `mp` (since it is the only write to that address in the litmus test). Thus, the load of `x` must have a value of 1 for this case, and RTLCheck encodes this information as a load value constraint for the right branch.

Once the constraint tree is generated, RTLCheck translates a given axiom to an SVA assertion as follows. `AND`, `OR`, and `NOT` nodes are translated to their SVA equivalents `and`, `or`, and `not`. Each μhb edge and node are translated according to the procedures in Sections 3.6.3 and 3.6.4 respectively. The load value constraints that apply to a given μhb edge or node are those that are ANDed to it. Thus, the constraint that the load of `x` return 0 in the left branch of the top-level `OR` applies to the edge `Ld x@WB`$\xrightarrow{hb}$`St x@WB`, but not to any other edge. Meanwhile, the constraint that the load of `x` return 1 in the right branch of the top-level OR applies to all other edges in the constraint tree.

3.6.3 Mapping Individual μhb Edges to SVA

When mapping a μhb edge `src`$\xrightarrow{hb}$`dest` to SVA, RTLCheck requires some notion of what the nodes `src` and `dest` represent at RTL. This functionality is provided by the node mapping function written by the user and provided to RTLCheck as input. Figure 3.12 shows pseudocode for a node mapping function for Figure 3.1's Multi-V-scale processor. The input parameters for a node mapping function are (i)

[13] Enforced through the `SameAddress` predicate in `NoInterveningWrite` in Figure 3.8.

```
fun mapNode(node, context, load_constr) :=
  let pc := getPC(node.instr, context) in
  let core := node.core in
  match node.stage with
  | IF => return "core[" + core + "].PC_IF == "
            + pc + " && ~stall_IF"
  | DX => return "core[" + core + "].PC_DX == "
            + pc + " && ~stall_DX"
  | WB => let lc := get_lc(node, load_constr) in
           let str := "core[" + core + "].PC_WB == "
                 + pc + " && ~stall_WB" in
           if lc != None then
             str += (" && load_data_WB == " + lc.value)
           return str
```

Figure 3.12: Multi-V-scale node mapping function pseudocode.

```
assert property (@(posedge clk) first |->
  (((((~((core[1].PC_WB == 32'd28 && ~(core[1].stall_WB)) ||
    (core[0].PC_WB == 32'd4 && ~(core[0].stall_WB)))) [*0:$]
    ##1
    (core[1].PC_WB == 32'd28 && ~(core[1].stall_WB) &&
    core[1].load_data_WB == 32'd0)
    ##1
    (~((core[1].PC_WB == 32'd28 && ~(core[1].stall_WB)) ||
    (core[0].PC_WB == 32'd4 && ~(core[0].stall_WB)))) [*0:$]
    ##1
    (core[0].PC_WB == 32'd4 && ~(core[0].stall_WB)))))
);
```

Figure 3.13: SV assertion checking Ld x@WB $\xrightarrow{hb}$ St x@WB in Multi-V-scale for mp where Ld x returns 0. The first bold red Verilog expression corresponds to mapping a node with a load value constraint (namely that Ld x is in WB and returns 0). The second bold red Verilog expression corresponds to mapping a node without a load value constraint (namely that St x is in WB). Some signal structure is omitted for brevity.

the node to be mapped, which is a specific microarchitectural event for a specific instruction, (ii) context information, such as the starting program counter (PC) for each core, and (iii) a list of load value constraints that must be obeyed by the edge being mapped (as described in Sections 3.5.4 and 3.6.2).

The output of the node mapping function is a Verilog expression that corresponds to the occurrence of the node to be mapped in RTL. The Verilog expression must contain a unique identifier for the instruction whose node is being mapped, so as to differentiate the mappings of two instructions for the same μhb node. If the node being mapped belongs to a load instruction, and it matches any of the load value constraints passed to the mapping function, the Verilog expression generated by the mapping function must also stipulate that the load has the required value at that point. For instance, in Figure 3.13, the first Verilog expression highlighted in bold red corresponds to mapping `Ld x@WB` from `mp` with the load value constraint that the load must return 0. Meanwhile, the second Verilog expression highlighted in bold red corresponds to mapping `St x@WB` from `mp` with no load value constraints. The value of `PC_WB` is used as a unique per-instruction identifier in the mappings.

The mapping of a μhb edge to SVA must be able to handle arbitrary delays before and between the `src` and `dest` nodes. An ordering edge $\texttt{src}\xrightarrow{hb}\texttt{dest}$ is a statement that `src` happens before `dest` for the execution in question. It says nothing about when `src` and `dest` occur in relation to other events in the execution, nor does it specify the *duration* of the delay between the occurrences of `src` and `dest`. A naive translation of such an edge to SVA would be to use the mechanism of delay ranges (Section 3.4.4):

```
##[0:$] mapNode(src, lc) ##[1:$] mapNode(dest, lc)
```

Here, `mapNode` represents the mapping function and `lc` the list of load value constraints. This SVA sequence allows an initial delay of 0 or more cycles (`##[0:$]`) before the occurrence of `src`, since `src` may not occur at the first cycle in the execution. It also includes an intermediate delay of 1 or more cycles (`##[1:$]`) between `src` and `dest`, as the duration of the delay between `src` and `dest` is not specified by the microarchitectural model.

Unfortunately, this standard mechanism is insufficient for checking that `src` does indeed happen before `dest` for all executions examined. Consider in isolation the edge in blue from Figure 3.8 in `BeforeAllWrites` enforcing that $\texttt{Ld x @WB} \xrightarrow{hb} \texttt{St x @WB}$, with a constraint that the load of `x` must return 0. At the same time, consider the execution trace of `mp` in Figure 3.9 which reflects the outcome where `St x @WB` $\xrightarrow{hb}$ `Ld x @WB` and the load returns 1. (The relevant signal values are underlined and in red in Figure 3.9.) Since Figure 3.9's execution has the events occurring in the opposite order and the load values are different, Figure 3.9 should serve as a counterexample to the property checking the edge from `BeforeAllWrites`. However, if one simply uses the straightforward mapping above (i.e. `##[0:$] <Ld x=0 @WB> ##[1:$] <St x@WB>`), Figure 3.9 is *not* a counterexample for the property!

The reason that Figure 3.9 is not a counterexample is that the unbounded ranges can match *any* clock cycle, including those which contain events of interest like the source and destination nodes of the edge. For this particular example, the initial `##[0:$]` can match the execution up to cycle 5. At cycle 6, since the load of `x` returns 1, `Ld x=0 @WB` does not occur, so the execution cannot match that portion of the sequence. However, nothing stops the initial delay `##[0:$]` from being extended another cycle and matching cycles 0-6. Indeed, even the entire execution can match `##[0:$]`, thus satisfying the property. Since Figure 3.9's execution never violates the sequence, it is not a counterexample to the property.

To address this problem of incorrect delay cycle matches, the conditions on the initial and intermediate delays must be stricter to stipulate that they are in fact repetitions of clock cycles where *no* events of interest occur. In this context, an event is of interest if it matches the node in question (i.e., if it matches the microarchitectural instruction and event), but *regardless of the data values themselves.* As such, for initial delays and intermediate cycles, RTLCheck uses this sequence:

```
(~(mapNode(src, None) || mapNode(dest, None))) [*0:$]
```

No load constraints are passed to the calls to the mapping function to generate the delay sequence. This prevents delay cycles from matching cases where events of interest occur with incorrect values, as in the case above for the load of `x` in `mp`. The overall translation scheme that RTLCheck uses for happens-before edges (henceforth referred to as *edgeTran*) is:

```
(~(mapNode(src, None) || mapNode(dest, None))) [*0:$]
##1 mapNode(src, lc) && ~mapNode(dest, None) ##1
(~(mapNode(src, None) || mapNode(dest, None))) [*0:$]
##1 mapNode(dest, lc)
```

This edge translation scheme is capable of handling variable delays while still correctly checking the edge's ordering. A simple proof of the edge translation scheme is provided below.

Theorem 1 (Edge Translation Correctness)**.** For any μhb edge $e = src \xrightarrow{hb} dest$, where *src* and *dest* represent unique events in microarchitectural and RTL execution, *edgeTran*(e) generates an SVA property that will return a counterexample if any RTL execution exists where both *src* and *dest* occur, but *src* does not happen before *dest*.

Proof. Any RTL execution *RExec* where both *src* and *dest* occur but *src* does not happen before *dest* must begin with the following: some number of cycles where $\neg src \wedge \neg dest$ holds, followed by a cycle where *dest* is true. In other words, *dest* becomes true before or at the same cycle where *src* becomes true, thus breaking the requirement that $src \xrightarrow{hb} dest$. This proof must show that *RExec* cannot match *edgeTran*(e).

Let *delayRange* be the SVA fragment `(~(mapNode(src, None) || mapNode(dest, None))) [*0:$]`, and let *srcTran* be the SVA fragment `mapNode(src, lc) & ~mapNode(dest, None)`. If *RExec* were to match *edgeTran*(e), the initial cycles where $\neg src \wedge \neg dest$ holds must match the initial *delayRange* in *edgeTran*(e), as they cannot match its subsequent *srcTran* portion. The subsequent cycle in *RExec* where *dest* becomes true cannot match the initial *delayRange*, since it requires that *dest* not hold (regardless of its value, if it is a load). Thus, this cycle must match *srcTran* for $RExec$ to match $edgeTran(e)$. However, this cycle cannot match $srcTran$, since $srcTran$ also requires that $dest$ not hold (again, regardless of its value, if it is a load). Thus, $RExec$ cannot match $edgeTran(e)$. ■

3.6.4 Mapping Node Existence Checks to SVA

Some properties require simply checking for the *existence* of a μhb node (i.e. `NodeExists <node>`), rather than an edge between two nodes. In these cases, we use a similar but simpler strategy to that used for mapping edges. The existence of a node is equivalent to a trace consisting of zero or more cycles where the node does not occur followed by a cycle where it *does* occur. However, simply translating a `NodeExists <node>` predicate using

```
(~mapNode(node, None)) [*0:$] ##1 mapNode(node, lc)
```

does not result in a property which ensures that `node` exists in the trace. This is because under SVA semantics, the possibly infinite repetitions of `~mapNode(node, None)` can match an entire trace where `node` never occurs, and the property is deemed to hold [IEE13, EF18]. Thus, this translation scheme would result in false negatives i.e. the property holding but RTL not actually guaranteeing the node's existence.

To ensure that the node exists in a trace, a trace must be required to match the entire sequence above. In other words, `mapNode(node, lc)` *must* occur at some cycle of the trace. SVA provides a method to do this through `strong` sequences, as opposed to the "weak" sequences discussed so far. A trace matches a strong sequence if and only if the trace matches the entire sequence, which is exactly what is required to ensure the existence of the node being checked for. (The semantics of strong sequences are analogous to those of the LTL **U** operator [EF18].)

Thus, RTLCheck translates a `NodeExists <node>` predicate to

```
strong((~mapNode(node, None)) [*0:$] ##1 mapNode(node, lc))
```

One might wonder why RTLCheck does not use strong sequences when translating an edge $src \xrightarrow{hb} dest$. Doing so would not just check for reorderings of *src* and *dest*, but also ensure that *src* and *dest* always eventually exist in a trace. The reason RTLCheck does not use strong sequences for edges is that they significantly increase runtime. For instance, if μhb edges are translated using the strong version of the sequence from *edgeTran*, JasperGold can only completely prove one of the generated assertions for `mp` on Multi-V-scale in 7 hours of runtime. On the other hand, if the weak sequences from *edgeTran* are used, JasperGold can prove all the generated properties for `mp` on Multi-V-scale in under 8 minutes. (See Section 3.9 for details.) RTLCheck's main function is to detect reorderings of events relevant to MCM verification, and weak sequences suffice if translating edges for this purpose. RTLCheck is not intended to verify properties related to deadlock, livelock, starvation, etc. The verification of such properties would need to ensure that specific events (like *src* and *dest*) exist in an execution.

3.6.5 Filtering Match Attempts

Consider the following SVA assertion:

```
p3: assert property (@(posedge clk) ##3 <St x@WB>);
```

If checking this assertion with respect to the execution in Fig. 3.9, one would expect the property to return true, as the WB stage of the store to `x` indeed occurs three cycles after the beginning of the execution. (Figure 3.9 elides the execution's first two cycles for brevity.) However, as Section 3.4.4 mentioned, SVA verifiers will check this assertion not just on a trace of the overall RTL execution, but on all suffixes of the trace as well. In other words, one match attempt of the property begins at cycle 0 and checks for `St x @WB` at cycle 3. Another match attempt begins at cycle 1 and checks for `St x @WB` at cycle 4, and so on for every cycle in the execution. If *any* of these match attempts fail, the entire property is considered to have failed. Here, the match attempt that begins at cycle 1 will fail, as `St x` will not be in `WB` at cycle 4 as the property requires.

Each μspec axiom corresponds to a microarchitecture-level ordering which is enforced once with respect to an execution as a whole. To achieve such semantics, the properties generated by RTLCheck must explicitly filter out match attempts that do not start at the beginning of RTL execution. To filter out match attempts other than the first attempt, RTLCheck guards each of its assertions with implications[14] triggered by the `first` signal, which is auto-generated by the Assumption Generator (Section 3.6.1). For instance, the guarded version of assertion `p3` above is:

```
assert property (@(posedge clk) first |->
    ##2 <St x@WB>);
```

Now, any match attempt that begins at a cycle after cycle 0 will trivially evaluate to true, as the `first` signal will be 0 and the implication consequent will never be evaluated. Meanwhile, the match attempt beginning at cycle 0 will trigger the implication and cause evaluation of the consequent property as required.

Putting it all together, Figure 3.13 shows an example assertion for `mp` which checks for the existence of an edge `Ld x=0 @WB` $\xrightarrow{hb}$ `St x @WB` on Multi-V-scale. As Section 3.6.3 explains, the mapping of the load's WB stage checks that the data value it returns is 0, as required by its load value constraint. In general, assertions check multiple edges, and are larger than the one in Figure 3.13.

3.7 Case Study: Multi-V-scale

This section describes the relevant details of the Multi-V-scale processor, a multicore version of the RISC-V V-scale processor [RIS15]. The V-scale processor itself is a

[14]The same functionality could be achieved by placing the assertion in a Verilog `initial` block [CDH+15].

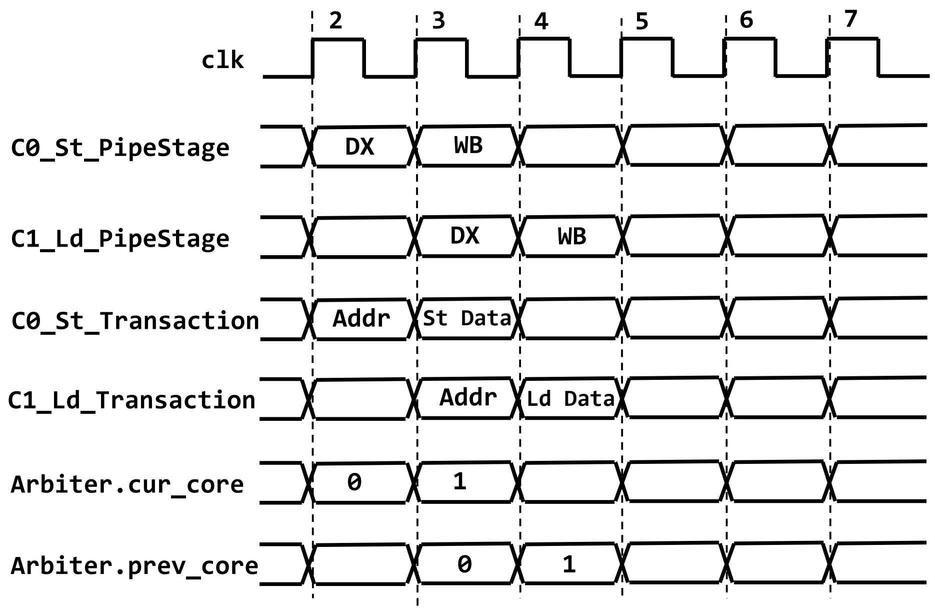


Figure 3.14: An example timing diagram for Multi-V-scale showing how a store on core 0 and a load on core 1 access memory through the arbiter in a pipelined manner.

Verilog implementation of the RISC-V Z-scale processor [LOM15], which is written in Chisel [BVR$^+$12].

3.7.1 V-scale Microarchitecture

The V-scale pipeline is suited for microcontrollers and embedded systems. It is similar in spirit to the ARM Cortex-M0/M3/M4 architectures [LOM15]. The V-scale pipeline is three stages long, as shown in Figure 3.1. The three stages are Fetch (IF), a combined Decode-Execute stage (DX), and the Writeback stage (WB).

The source code of V-scale does not implement a cache, but does have an implementation of a memory array. When accessing memory, both loads and stores send their addresses to memory in the DX stage. In the WB stage, a load receives its data from memory, and a store provides its data to memory, to be clocked in on the next rising edge.

The V-scale memory is pipelined, and is able to start a memory transaction every cycle. Thus, it can start a memory transaction for an instruction `a` which is in DX while providing data to or reading data from an instruction `b` which is in WB. Figure 3.14 shows a timing diagram of how V-scale loads and stores operate. The memory does not always operate as expected; RTLCheck discovered a bug in its implementation in the course of its analysis. (See Section 3.9.1.)

3.7.2 Multi-V-scale

I created a multicore version of the V-scale processor by instantiating four V-scale pipelines and connecting them to data memory through a simple arbiter that I developed (Figure 3.1). The arbiter is connected to all four cores, and allows only one of them to access data memory every cycle. If cores other than the one currently granted access wish to access memory, they must stall in the DX stage until the arbiter grants them access.

The arbiter is capable of switching from any core to any other core in any cycle. The switching pattern of the arbiter is dictated by a top-level input to the design. This input stipulates which core the arbiter should grant memory access to in the next cycle. JasperGold tries all possibilities for this input, resulting in all possible switching scenarios between cores being examined when verifying properties generated by RTLCheck. The arbiter accounts for the pipelined nature of the V-scale memory; in other words, it can allow one core to start a request to memory in its DX stage while another core is receiving data from or sending data to memory in its WB stage. Figure 3.14 shows an example of this pipelining.

In addition to making V-scale multicore, I also added halt logic and a halt instruction to the V-scale implementation. This halt logic lets a thread be stopped once it has executed its instructions for a litmus test. (At the time, the RISC-V ISA did not have a halt instruction that I could have used.)

3.7.3 Modelling Multi-V-scale in μspec

I modelled our Multi-V-scale processor in μspec by having one node per instruction per pipeline stage (i.e. one each for IF, DX, and WB respectively). I included Figure 3.8's axiom, which states that loads must read from the last store to their address to complete its WB stage, or from the initial state of memory. This axiom should hold since stores write to and loads read from the same memory. I also included an axiom enforcing a total order on the DX stages of all memory instructions. This axiom is enforced by the arbiter allowing only one core to access memory at a time while the others stall in DX. I also included properties such as the one in Figure 3.3b stating that the pipeline stages were in-order. Another axiom I added required a total order on all writes to the same address (enforced through the arbiter's total order on memory operations). Figure 3.3a depicts an example μhb graph showcasing the edges added by some of these axioms.

3.8 RTLCheck Methodology and Usage Flows

RTLCheck's Assertion Generator and Assumption Generator are written in Gallina, the functional programming language of the Coq proof assistant [Coq04]. As with PipeCheck, I use Coq's capability to be extracted to OCaml to generate an OCaml version of RTLCheck that can be compiled and run as a standalone binary.

RTLCheck's generated assertions and assumptions are output as a single file per litmus test, taking only a few seconds per test. A shell script uses these files to create litmus-test-specific top-level modules of Multi-V-scale which are comprised of the implementation of the top-level module concatenated with all the assertions and assumptions for that specific litmus test. I based my changes to V-scale to make it multicore off commit `350c262` in the V-scale git repository [Mag16].

3.8.1 RTLCheck Methodology

RTLCheck uses JasperGold [Cad15b] to verify the SV assertions subject to the SV assumptions for a given litmus test. JasperGold compiles the RTL implementation and any SVA property to be proven on the RTL into finite automata, with state transitions at clock cycle boundaries. A property is valid with respect to the RTL if all execution traces that can be generated by the RTL satisfy the property [CDH+15]. For each property, JasperGold may: (i) prove it for all possible cases, (ii) find a counterexample, or (iii) prove it for all traces shorter than a specified number of cycles (bounded proof).

JasperGold has a variety of different proof engines which are tailored to different purposes. These engines employ SAT (satisfiabililty) and BDD (binary decision diagram)-based approaches to prove the correctness of properties [Cad16]. Section 3.9 discusses my findings regarding the suitability of various engines for RTLCheck.

When verifying properties, JasperGold takes in a Tcl script as its configuration, which includes the choice of engines to use, how long to allot for the overall verification, and how often to switch between properties when verifying. I use a shell script to generate a Tcl script from a template for each litmus test. The Tcl scripts include a reference to the top-level module for their specific litmus test in the files provided to JasperGold for verification. The test-specific scripts and top-level modules allow instances of JasperGold to be run in parallel for different tests on the same Verilog design without duplication of most of the Verilog RTL. I ran instances of JasperGold on the Multi-V-scale design across litmus tests on a 224-node Intel cluster, allotting 5 cores and 64-120 GB of memory per litmus test (depending on the configuration used). I use a suite of 56 litmus tests, comprised of a combination of hand-written tests from the x86-TSO suite [OSS09] and tests automatically generated using the `diy` framework [diy12]. Section 3.9 gives verification results.

3.8.2 RTLCheck Usage Flows

As Section 3.1 mentions, RTLCheck can be used in one of two ways. If the RTL for a processor exists and a user is creating a μspec model for this existing processor, RTLCheck can be used to verify the soundness of the μspec model with respect to the RTL for suites of litmus tests. On the other hand, if the user created a μspec model of their processor prior to writing its RTL, RTLCheck can be used to verify that the eventual RTL maintains the axioms of that μspec model for suites for litmus tests.

RTLCheck's operation is exactly the same for both use cases; the only difference is how the user should interpret results from the SVA verifier. If verifying soundness, the RTL specifies the desired behaviour and the μspec axioms must correspond to this behaviour. Thus, if JasperGold returns a counterexample to a generated SVA property in this use case, the user must modify their μspec axioms to correctly correspond to the RTL. On the other hand, if verifying RTL, the μspec axioms specify the desired behaviour and RTL must maintain the orderings specified by these axioms. So if JasperGold returns a counterexample to an SVA property in this scenario, the user should modify their RTL to correctly maintain the orderings specified by the μspec axioms.

3.9 Results

This section presents the results of my evaluation of the Multi-V-scale processor RTL. I first discuss a bug I found, and then present RTLCheck runtimes under different JasperGold configurations to compare engines.

3.9.1 Bug Discovered in the V-scale Processor

In my evaluation of Multi-V-scale, JasperGold reported a counterexample for a property verifying the `Read_Values` axiom (Figure 3.8) for the `mp` litmus test. This

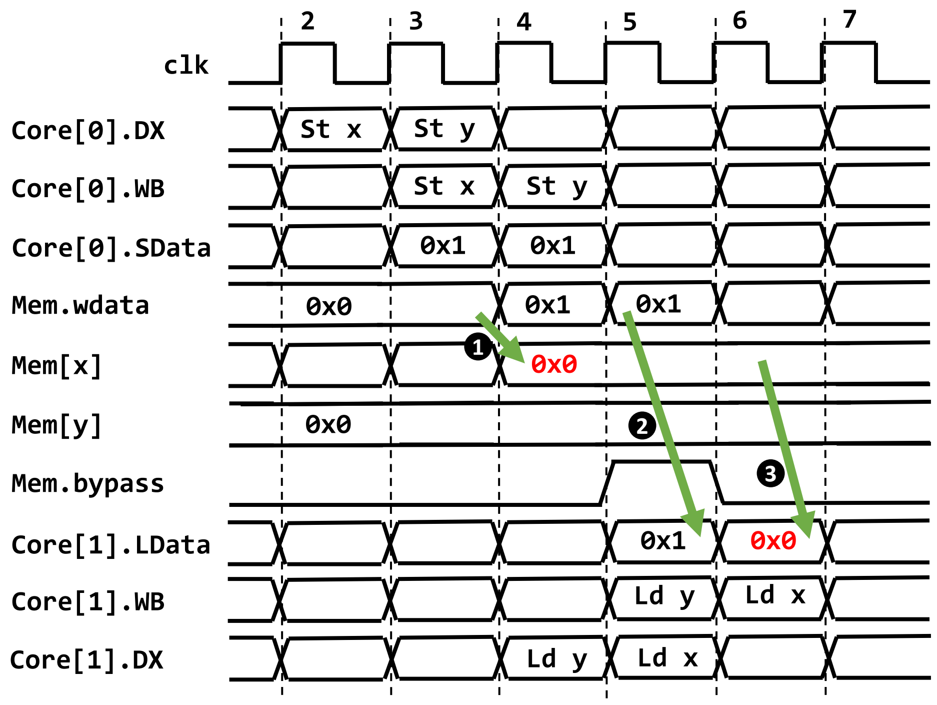


Figure 3.15: An execution of `mp` showcasing the bug RTLCheck found in the memory implementation of V-scale. The store to `x` is dropped by the memory, resulting in the subsequent load of `x` returning an incorrect value of 0.

property checks that each read returns the value of the last store to that address that completed `WB` (provided the read did not occur prior to all writes). Investigating the counterexample trace, I discovered a bug in the memory implementation of the V-scale processor. Namely, if two stores are sent to memory in successive cycles, the first of the two stores is *dropped* by the memory. The V-scale memory presents a ready signal to the pipeline (or in the multicore case, to the arbiter), and the implementation currently hard-codes this signal to be high. This hard-coded value implies that the memory is ready to accept a new value to be stored every cycle, and so the dropping of values is a bug. This bug would occur even in a single-core V-scale.

Internally, the memory clocks data from stores into a register `wdata`, and only writes the contents of `wdata` to the address of the store in memory when another store initiates a transaction. If a load requests data from the address whose latest store is in `wdata`, the data is bypassed to the load by the memory. Thus, `wdata` functions in a manner akin to a single-entry store buffer.

Figure 3.15 shows a counterexample trace which violates the RTLCheck-generated property. Here, the two stores of `mp` (to `x` and `y`) initiate memory transactions in cycles 2 and 3 respectively. The `wdata` register is consequently updated to 1 in cycles 4 and 5, one cycle after the stores provide their data values. However, when the second store initiates its transaction at cycle 3, the memory implementation incorrectly pushes the value of `wdata` from cycle 3 into memory at address `x` (arrow ❶ in Figure 3.15) to make room in `wdata` for the store of `y`. At cycle 3, `wdata` has not yet been updated with the data of the store to `x`, so `x` gets incorrectly set to 0 in memory. The data of the store to `y` is then clocked into `wdata` at the start of cycle 5, overwriting the data of the store to `x` and causing it to be lost. When the load of `y` occurs, it gets its value bypassed from `wdata` (arrow ❷ in Figure 3.15). This is because no subsequent store has occurred to push the contents of `wdata` to memory. Meanwhile, the load of `x` returns the value of memory at address `x` (arrow ❸ in Figure 3.15), which is incorrectly 0, violating the property.

I corrected the dropping of stores by eliminating the intermediate `wdata` register. Instead, I clock a store's data directly into memory one cycle after the store does its WB stage. Load data is combinationally returned in WB as the value of memory at the address the load is accessing. This organization allows data written by a store in one cycle to be read by a load in the next cycle. Once I fixed the bug, JasperGold was able to completely prove or provide a bounded proof for all RTLCheck's generated assertions. This bug was also independently reported [com16], but that report does not correctly diagnose the bug as only occurring upon successive stores. RTLCheck's counterexample trace offered detailed analysis to pinpoint the bug's root cause.

This example highlights an interesting and important use case for RTLCheck: it is most directly intended to catch memory ordering bugs, but it is also able to catch bugs that fall on the boundary between memory consistency bugs and more basic module correctness issues. This is because formal verification of RTL takes into account all signals that may affect the property being checked, whether they are explicitly modeled at the microarchitectural level or not. Thus, *any* behavior that causes the property to be invalidated for a litmus test will be flagged as a counterexample by a property verifier checking assertions generated by RTLCheck.

3.9.2 RTLCheck Runtimes

I ran the properties generated by RTLCheck under the JasperGold commercial RTL verifier for the 56 litmus tests in the suite I used. JasperGold has many configuration options and solver settings. The size of the configuration space precludes an exhaustive discussion, but I present results for two options.

Table 3.1 provides the details of the configurations. Each configuration spends one hour trying to verify tests by finding covering traces for assumptions (see Section 3.6.1), and then runs proof engines on the assertions for the remaining 10 hours. The Hybrid configuration uses a combination of bounded engines (which aim to prove correctness up to a bounded number of cycles) and engines which aim to find full proofs, and also utilizes JasperGold's autoprover. The second configuration (Full_Proof) uses exclusively full proof engines. These engines can theoretically improve the percentage of proven properties, potentially at the cost of increased runtime.

Figure 3.16 presents the runtime to verification of JasperGold for the 56 litmus tests in the suite. For the 23 tests where assumptions were proven to be unreachable through covering traces, the runtime to verification is simply the time taken to check the assumptions. This time can be quite small, as seen for tests like `lb`, `mp`, `n4`, `n5`, and `safe006`, which are verified in under 4 minutes by either configuration. For tests where assumptions were not found to be unreachable, the total runtime is either the time taken to prove all properties, or the maximum runtime of 11 hours per test (if

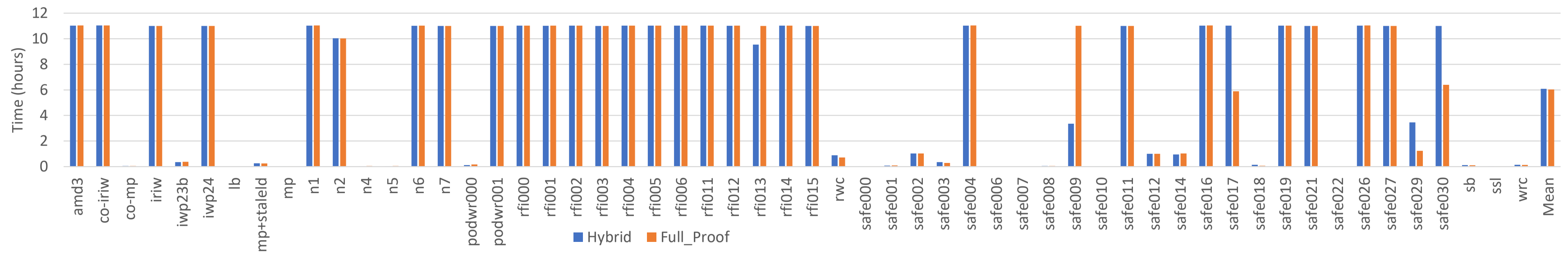


Figure 3.16: JasperGold runtime for test verification across all 56 tests and both engine configurations.

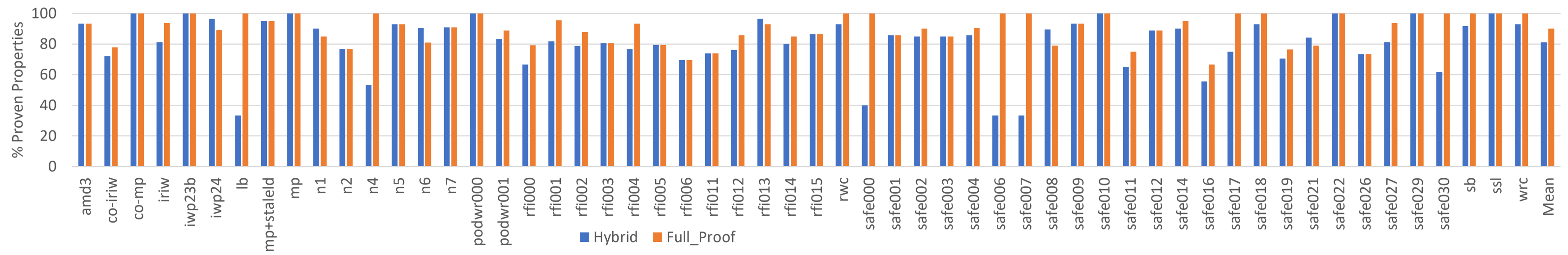


Figure 3.17: Percentage of fully proven properties (in a max. of 11 hours) across all 56 tests and both engine configurations.

Config	**Covering Trace Run**	**Proof Engine Runs**	**Memory/ Test**	**Cores/ Test**
Hybrid	1 hour	Autoprover (1 hr) K I N AM AD (9 hrs)	64 GB	5
Full_Proof	1 hour	I N AM AD (10 hrs)	120 GB	5

Table 3.1: JasperGold configurations used when verifying Multi-V-scale with RTLCheck.

some properties remained unproven). The average runtime is 6.1 hours per test for the Hybrid configuration and 6.0 hours per test for the Full_Proof configuration. The total CPU time for the Hybrid run is 2080 hours, and that of the Full_Proof run is 2138 hours. Both runs used 5 threads per test.

Figure 3.17 shows the percentage of all assertions generated by RTLCheck that JasperGold was able to find complete proofs for within the time limits provided for the 56 litmus tests. In most cases, the Full_Proof configuration can find complete proofs for an equivalent or higher number of properties than the Hybrid configuration can. However, there are tests where the Hybrid configuration does better, such as `n1`, `n6`, and `rfi013`. On average, the Hybrid configuration was able to completely prove 81% of the properties per test, while the Full_Proof configuration found complete proofs for 90% of the properties per test. Overall, the Hybrid configuration found complete proofs for 81% of all properties, while the Full_Proof configuration found complete proofs for 89% of them. Given that the average runtime of the Full_Proof configuration is slightly lower than that of the Hybrid configuration, using exclusively full proof engines has clear benefits as it can find complete proofs for a larger fraction of the properties.

For properties that were not completely proven, JasperGold provides bounded proofs instead. The average bounds for such properties for the Hybrid and Full_Proof configurations were 44 and 23 cycles respectively. As litmus tests are relatively short programs, many executions of interest fall within these bounds, giving the user considerable confidence that the implementation is correct.

3.10 Related Work on Formal RTL Verification

Aagaard et al. [ACDJ01] propose a framework for microprocessor correctness statements that compares specification and implementation state machines. They also propose a formal verification methodology [AJM$^+$00] for datapath-dominated hardware using a combination of lightweight theorem proving and model checking, integrated within the FL functional language.

There has been work in the CAD/CAV communities on assertion-based verification (ABV) [TS08, YL05]. However, there is no prior work (to my knowledge) on using such assertions for multicore MCM verification. In addition, such work focuses on handwritten assertions, in contrast to RTLCheck's automatic assertion and assumption generation. In that regard, RTLCheck is closer to the ISA-Formal verification methodology created by Reid et al. [RCD$^+$16] to verify ARM microprocessors. They use a processor-specific abstraction function (similar to RTLCheck's node mapping function) which extracts architectural state from microarchitectural state, and check correctness by comparing the difference in the architectural state before and after an instruction executes/commits with what a machine-readable specification says the instruction should do to architectural state. They do not verify the entire design. For instance, the memory subsystem and floating-point units are not verified, and they do not address memory consistency issues.

Pellauer et al. [PLBN05] provide a method for synthesizing SVA into finite state machine hardware modules, which can then check for the desired behavior as the processor runs. Stewart et al. [SGNR14] proposed DOGReL, a language for specifying directed observer graphs (DOGs). These DOGs describe finite state machines of memory system transaction behavior. Users also define an *abstractor* per interface (similar to RTLCheck's node mapping function) that interprets signal-level activity as transaction-level events, whose properties can then be verified. DOGReL compiles down to RTL and SVA, similar to RTLCheck. However, RTLCheck specifically focuses on MCM properties in multiprocessors, which are not discussed in the DOGReL paper. In addition, RTLCheck's μhb graphs represent executions while DOGs represent finite state machines.

Kami [VCAD15, CVS+17] (Section 2.3.4) enables users to create processor implementations based on labelled transition systems (LTSes) and then manually prove the MCM correctness of such processors across all programs. Kami includes an automated process for extracting a Bluespec hardware implementation from such a proven-correct Kami implementation. To date, Kami proofs of MCM correctness have only been demonstrated for SC. While RTLCheck proves correctness for litmus tests rather than all programs, its methodology is capable of handling arbitrary RTL designs that may implement a variety of ISA-level memory models, not just SC. Furthermore, RTLCheck does not require complicated manual proofs like Kami does, and RTLCheck does not require the design to be written in Bluespec to be synthesizable.

3.11 Chapter Summary

The microarchitectural MCM verification conducted by tools like PipeCheck [LPM14, LSMB16] and PipeProof (Chapter 5) is adept at discovering MCM bugs in early-stage design ordering specifications. Similarly, the `herd` framework [AMT14] enabled the formal specification and automated formal analysis of ISA-level MCMs (Section 2.3.1). However, the prior disconnection of such techniques from RTL verification impeded the ability of such tools to ensure the correctness of taped-out chips. If constructing a μspec model for an existing processor, there was no way to formally check whether the μspec ordering specification was a sound representation of the underlying RTL. Similarly, if writing RTL for a design whose orderings had previously been specified as a μspec model, there was no way to formally check whether the RTL matched the ordering requirements of the μspec axioms. Both of these use cases require a way to translate μspec axioms to equivalent RTL properties.

This chapter addresses the above issues by developing RTLCheck, a methodology and tool for the automatic translation (given appropriate mapping functions) of μspec axioms to SVA assertions and assumptions for suites of litmus tests. RTLCheck bridges the gap between the starkly different logics and semantics of μspec and SVA to accomplish its translation. RTLCheck also works around an over-approximation used by SVA verifiers that provides improved performance but makes the translation of μspec to SVA harder. The assertions generated by RTLCheck can be formally verified against an RTL implementation using commercial verifiers like Cadence JasperGold, giving architects and engineers confidence that their RTL and μspec axioms are in agreement. In combination with PipeProof (Chapter 5) which links `herd` ISA-level MCMs to μspec models, RTLCheck also enables the formal linkage of `herd` ISA-level MCMs to Verilog RTL.

RTLCheck's translation enables the MCM verification of processor implementations written in Verilog RTL. Its translation also enables the soundness verification (on a per-test basis) of microarchitectural specifications with respect to the processor implementations that they model. RTLCheck's linkage of early-stage design ordering specifications to that of RTL implementations helps propagate the early-stage correctness guarantees of tools like PipeCheck and PipeProof to the taped-out chips that are eventually shipped to end users. Furthermore, it enables the post-implementation verification required by a progressive verification flow (Chapter 6) for MCM properties in parallel processors. RTLCheck is open-source (apart from the commercial JasperGold verifier) and is available at github.com/ymanerka/rtlcheck.

Chapter 4

Scalable MCM Verification Through Modularity[1]

The empire, long divided, must unite; long united, must divide. Thus it has ever been.
—Luo Guanzhong (tr. Moss Roberts)
Three Kingdoms

The ISA-level MCM of a processor is enforced by a combination of the processor's individual components, e.g. its caches, store buffers, pipelines, etc. As such, it makes intuitive sense to conduct hardware MCM verification by reasoning about the orderings enforced by various processor components all at once. However, modern processors are incredibly complex heterogeneous parallel systems, consisting of a vast number of components. Meanwhile, the SAT and SMT solvers used by automated MCM verification approaches like PipeCheck are NP-complete [Coo71], so once the query provided to the solver grows beyond a certain size, runtime and/or memory usage tends to explode. As such, the monolithic verification of approaches like PipeCheck does not scale to detailed models of commercial designs.

[1]An earlier version of the work in this chapter was previously published on arXiv [MLM20]. I was first author on the publication.

A related challenge in the MCM verification of commercial designs is that each of the components in such a design may be developed by a distinct team. This makes it hard to write a monolithic specification detailing the orderings enforced by each of the components, as PipeCheck requires. Ideally, ordering specification would be *modular*, i.e., each team would be able to specify the orderings enforced by their components independently and then connect them together when conducting MCM verification.

To solve the twin challenges of scalability and specification modularity in hardware MCM verification, this chapter presents RealityCheck, a methodology and tool for automated formal MCM verification of modular microarchitectural ordering specifications. RealityCheck allows users to specify their designs as a hierarchy of distinct modules connected to each other rather than a single flat specification. It can then automatically verify litmus test programs against these modular specifications. Most importantly, RealityCheck supports abstracting design modules using interface specifications. This enables scalable verification by breaking up the verification of the entire design into smaller verification problems.

4.1 Introduction

Microprocessors today are complex heterogeneous systems developed by many individuals. Processor development is divided up among different teams, with each team responsible for one or a few components. For instance, one team may be responsible for the pipeline, another for the store buffer, a third for the L1 caches, and a fourth for an accelerator. At the System-on-Chip (SoC) level, components may even be developed by different companies. Nevertheless, a processor or SoC created by the interconnection of various components developed by different teams and companies must function correctly.

The conformance of a processor to its MCM is one measure of its correctness. In prior work, PipeCheck [LPM14, LSMB16] (Section 2.4) enabled automated microarchitectural MCM verification for the first time, using model checking (Section 2.2.1) backed by a custom SMT solver (Section 2.2.1). However, hardware designs are modelled in PipeCheck as a single monolithic specification consisting of all the individual smaller orderings enforced by the processor's various hardware components. Such monolithic verification will not scale to detailed models of large designs like those of today's commercial processors due to the NP-completeness of the SAT and SMT solvers used by such approaches [Coo71].

PipeCheck's monolithic verification also puts it at odds with the realities of processor development. The distributed nature of the hardware design process makes it difficult for processor design teams to write a single monolithic specification for the entire processor, as PipeCheck requires. Ideally, microarchitectural ordering specifications would be *modular*, i.e. each team would be able to specify the orderings enforced by their components independently as distinct modules with clearly defined module boundaries. The individual modules would then be connected together when conducting MCM verification.

PipeCheck's monolithic verification also leads to a prevalence of *omniscient* or global properties in its design specifications. For example, a property guaranteed by most shared-memory systems today is that of coherence [SHW11]. At an instruction level, coherence requires that there exists a total order on all stores to the same address that is respected by all cores in the system. However, hardware implementations of coherence use *distributed* protocols where each cache (and often a bus/directory) is responsible for enforcing part of the orderings required. None of the hardware components in such an implementation has omniscient visibility of the entire processor, and none of them can make statements about the global behaviour of the system. Thus, omniscient properties such as the coherence definition above reflect a designer's

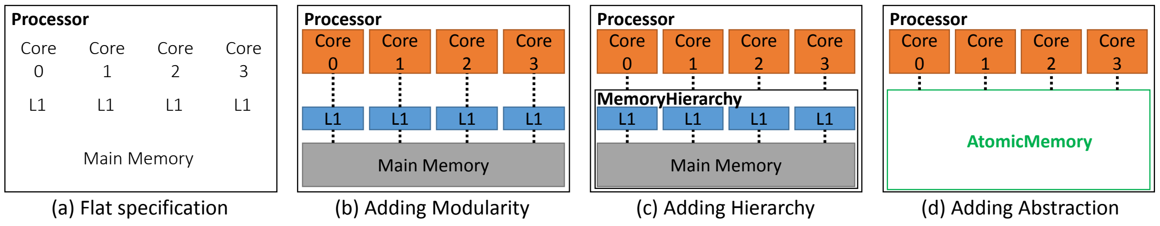


Figure 4.1: Illustration of adding modularity, hierarchy, and abstraction to a flat design specification. The flat design specification in (a) does not incorporate any of the structural modularity and hierarchy present in today's hardware designs. The addition of modularity in (b) groups individual design components into their own modules, breaking down the specification into multiple pieces. Hierarchy, shown in (c), enables larger modules like `MemoryHierarchy` to be built out of smaller modules. Finally, (d) demonstrates the use of abstraction by using an interface `AtomicMemory` to abstract the module `MemoryHierarchy`. This decouples the specification of `MemoryHieарchy`'s implementation from the specification of its external behaviour.

high-level view of the hardware rather than what the hardware is actually doing. If the designer's high-level view is inaccurate, verification using such a specification will be unsound.

As this chapter will show, each of the above problems can be solved by developing an automated microarchitectural MCM verification methodology that incorporates three well-known concepts: **modularity**, **hierarchy**, and **abstraction**. Figure 4.1 shows a graphical depiction of these concepts in relation to hardware MCM verification. PipeCheck provides *flat* verification (Figure 4.1a). There is no way for users to encapsulate component functionality into a unit whose properties only apply to that unit, or to verify a component independently of the rest of the system. In other words, there is no support for **modularity** as it is depicted in Figure 4.1b. Similarly, users could not build larger modules from smaller ones as there was no support for **hierarchy**. Figure 4.1c shows an example of hierarchy where the L1 and Main Memory modules reside within the `MemoryHierarchy` module.

PipeCheck also has no support for **abstraction**. In other words, there is no way for users to decouple the specification of their component's external behaviour from the specification of its implementation. As an example of abstraction, Figure 4.1d represents the `MemoryHierarchy` using an abstract interface `AtomicMemory`. `AtomicMemory` specifies the external-facing behaviour of the memory hierarchy, but says nothing about the internal implementation, like how many caches there are.

To enable scalable automated microarchitectural MCM verification that is in line with the realities of processor design, this chapter presents RealityCheck, a methodology and tool for automated formal MCM verification which supports modularity, hierarchy, and abstraction. RealityCheck allows users to specify their design as a hierarchy of distinct modules connected to each other (similar to an object-oriented programming language like C++), closely matching the structure of real hardware designs. RealityCheck can then automatically verify whether the composition of the various modules exhibits behaviours forbidden by the ISA-level MCM of the processor through bounded verification for suites of litmus test programs. RealityCheck also lets users write interface specifications of the external behaviour of components; it can then verify component implementation specifications against these interfaces up to a bound. The use of such abstraction in concert with modularity and hierarchy allows a design to be verified piece-by-piece, breaking down verification of the entire design into smaller verification problems. This allows automated microarchitectural MCM verification to scale to large designs like those of commercial processors. Finally, RealityCheck fulfills the requirement for detailed design verification in a progressive verification flow (Chapter 6) for MCM properties in parallel processors.

The only prior hardware MCM verification work that supports modularity, hierarchy, and abstraction is Kami [VCAD15, CVS+17] (Section 2.3.4). While Kami can prove MCM correctness across all possible programs (which RealityCheck cannot), these proofs must be written manually in a framework in the Coq proof assistant (Section 2.2.2). This requires significant manual effort and formal methods knowledge. Thus, Kami is unsuitable for use by typical computer architects, as they do not have

Core 0	**Core 1**
(i1) [x] ← 1 (i2) r1 ← [y]	(i3) [y] ← 1 (i4) r2 ← [x]
SC forbids r1=0, r2=0	

Figure 4.2: Code for litmus test `sb`

such formal methods expertise. In contrast, RealityCheck is an automated tool that is easy to use while still providing modularity, hierarchy, and abstraction.

The rest of this chapter is organised as follows. Section 4.2 uses a concrete example to illustrate the issues caused by PipeCheck's lack of modularity, hierarchy, and abstraction. Section 4.3 provides an overview of RealityCheck. Section 4.4 covers abstraction and its benefits in the context of RealityCheck, including how it enables scalable verification when used in combination with modularity and hierarchy. Section 4.5 covers μspec++, a domain-specific language developed as part of RealityCheck to enable the creation of hierarchical modular microarchitectural ordering specifications. μspec++ extends the μspec domain-specific language of PipeCheck [LSMB16] (Section 2.4.2) to incorporate modularity, hierarchy, and abstraction. Section 4.6 covers RealityCheck's operation, while Section 4.7 provides examples of the different ways in which RealityCheck can be used in hardware design flows. Section 4.8 covers RealityCheck's methodology and results, and Section 4.9 concludes.

4.2 Motivating Example

4.2.1 Flat Verification using PipeCheck

Figure 4.3 shows a μhb graph (Section 2.4.1) depicting the execution of the `sb` litmus test (Figure 4.2) on the microarchitecture represented by Figure 4.1b (henceforth called `exampleProc`). Assume that each core has 3-stage in-order pipelines of Fetch (IF), Execute (EX), and Writeback (WB) stages, and that the processor aims to implement SC. As a reminder, in `sb`, both addresses `x` and `y` are initially 0 by convention. Core 0

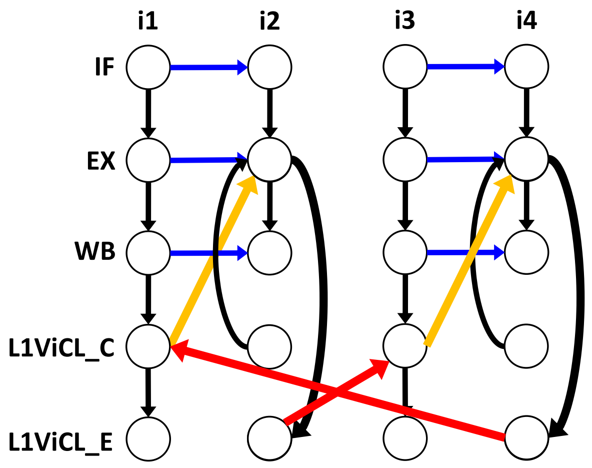


Figure 4.3: Example μhb graph for `sb` litmus test on Figure 4.1b's processor.

```
Axiom "Read_Initial":
forall microop "i", forall microop "j",
 IsAnyRead i /\ DataFromInitialState i /\
 IsAnyWrite j /\ SameAddress i j =>
   AddEdge((i,L1ViCL_E),(j,L1ViCL_C),"").
```

Figure 4.4: Example μspec axiom.

sets its flag `x` and reads the value of core 1's flag `y`. Meanwhile, core 1 sets its flag `y` and reads the value of core 0's flag `x`. Under SC, it is forbidden for both loads to return 0, as there is no total order on all memory operations that would allow this.

The last two rows in the μhb graph (`L1ViCL_C` and `L1ViCL_E`) use the ViCL (Value in Cache Lifetime) abstraction [MLPM15] to model cache occupancy and coherence protocol events relevant to MCM verification. Briefly speaking, a ViCL represents the period of time (relative to a single cache or main memory) over which a given cache/memory line provides a specific value for a specific address. The time period referenced by a ViCL begins at a *ViCL Create* event, and ends at a *ViCL Expire* event. A ViCL for a given address in a given cache is created when a cache line for that address is brought into the cache, or when the value of that address is updated in the cache. Similarly, a ViCL for a given address in a given cache expires when the cache line for that address is evicted, or when the value for that address is updated (which results in the creation of a new ViCL to represent the new value).

Figure 4.3 uses `L1ViCL_C` and `L1ViCL_E` to refer to ViCL Create and ViCL Expire events respectively, for the L1 caches in `exampleProc`. CCICheck [MLPM15] provides a formal definition of and further details on ViCLs.

Figure 4.4 shows a μspec axiom (Section 2.4.2) for `exampleProc` that specifies some of the orderings enforced by the coherence protocol (Section A.5). This axiom enforces that for every `microop i` (a `microop` is a single load or store), if it is a load (enforced through the `IsAnyRead` predicate) that reads from the initial state of memory (`DataFromInitialState i`), then its L1 ViCL must expire before the creation of L1 ViCLs of any write `j` to that address, as the write would have caused the invalidation of all other cache lines for that address. The loads `i2` and `i4` in `sb` both read from the initial state, so this axiom adds the red edges in Figure 4.3 to enforce the expiration of their ViCLs before the creation of ViCLs for stores `i3` and `i1` respectively.

4.2.2 Deficiencies of Flat Verification

Lack of Scalable Verification

μspec specifications that reason over a limited number of nodes and edges per instruction, such as the μspec specification for `exampleProc`, can be verified against a litmus test by PipeCheck in seconds or minutes [LSMB16]. However, commercial processors are far more complex. While PipeCheck has verified designs of commercial processors such as the OpenSparc T2 [LPM14] and a Sandy Bridge design [LSMB16], it did so using high-level organisational models that do not reflect the true complexity of these designs. These models also used omniscient axioms (explained below), which reflect architectural intent rather than what the design is actually doing.

If a user wished to use PipeCheck to verify a more detailed model of e.g., a Sandy Bridge design, the μspec for that model could well deal with hundreds of nodes and edges per instruction. PipeCheck provides no way to break down the verification of a processor into smaller parts, so its verification of such large μspec specifications would quickly become infeasible due to the NP-completeness of SMT solving [BSST09].

PipeCheck's monolithic verification provides no way for users to verify a component independently of the rest of the system. For instance, all verification in Figure 4.3's μhb graph for `exampleProc` is conducted in terms of instructions, relative to an ISA-level litmus test. There is no way to verify say, the L1 caches individually. Ideally, each design component would have an interface specification that it could be verified against, and it would be possible to verify a design piece-by-piece. This would allow model checking-based approaches for hardware MCM verification to scale to large detailed designs.

Lack of Modular Specifications

Writing a detailed μspec specification for a large processor in PipeCheck is itself difficult. PipeCheck provides no way for users to specify individual hardware components independently of the rest of the system. There is no way to encapsulate a set of μspec axioms and the nodes and edges they operate on into a single unit that can be instantiated and replicated elsewhere in the design (much like classes and objects in C++ or Verilog RTL modules). For instance, in the case of `exampleProc`, there is no way to specify an `L1Cache` module that details the orderings enforced by a given L1 cache. Instead, the user creates a single monolithic specification containing all the axioms relevant to the design.

PipeCheck's monolithic specifications will cause difficulties if PipeCheck is used in the design cycle of a commercial processor. This is because processors are designed by a number of distinct teams, with each team responsible for one or a few components. Each team will have detailed knowledge about their component, and will be well placed to write axioms for it. However, they may know very little about components designed by other teams. As PipeCheck specifications are monolithic, users have no way to specify module boundaries to clearly demarcate the connections between modules. This makes it hard to ensure that the axioms for a given team's component correctly interface with the axioms for another team's component. Such incorrect component interactions will result in flawed specifications, causing bugs to be missed (false negatives) or spurious verification failures (false positives). Ideally, each team would write the axioms for their component as a module with clearly defined boundaries, and these modules could be instantiated and connected together inside larger modules. The specification for the overall processor would be created by instantiating and connecting various modules together according to a specific hierarchy (much like how processors are written in Verilog RTL today).

Omniscient Axioms

PipeCheck's flat verification encourages the use of axioms which exercise an omniscient (or global) view of the entire processor. For instance, Figure 4.4's axiom adds μhb edges between ViCLs of Core 0's L1 and Core 1's L1 in Figure 4.3 to reflect that the loads must read memory before the stores invalidate their cache lines through the coherence protocol. While this axiom is straightforward to write and would be valid in a coherent memory hierarchy, it does not directly correspond to how the hardware actually works. The ordering in Figure 4.4's axiom is actually enforced in a distributed manner by a combination of modules, specifically the L1s and (likely) a bus or directory. Each component in a hardware design can only enforce orderings on the events that it sees. For example, in `exampleProc`, Core 0's L1 can observe the values in its own cache but it cannot see the values in Core 1's L1. An architect may surmise that the combination of the orderings enforced by the L1s and bus/directory is sufficient to enforce Figure 4.4's axiom, but such an assumption must be validated before using the axiom for microarchitectural MCM verification. Otherwise it is possible that the axiom does not hold in the actual hardware design, leading to unsound verification. A better

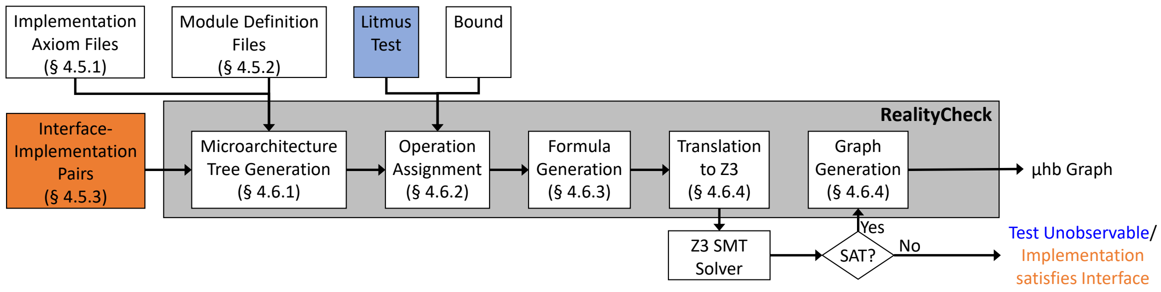

Implementation Axiom Files (§ 4.5.1)
Module Definition Files (§ 4.5.2)
Litmus Test
Bound
RealityCheck
Interface-Implementation Pairs (§ 4.5.3)
Microarchitecture Tree Generation (§ 4.6.1)
Operation Assignment (§ 4.6.2)
Formula Generation (§ 4.6.3)
Translation to Z3 (§ 4.6.4)
Graph Generation (§ 4.6.4)
µhb Graph
Z3 SMT Solver
SAT?
Yes
No
Test Unobservable/
Implementation satisfies Interface

Figure 4.5: RealityCheck block diagram. Orange parts only apply to interface verification, and blue parts only to litmus test verification.

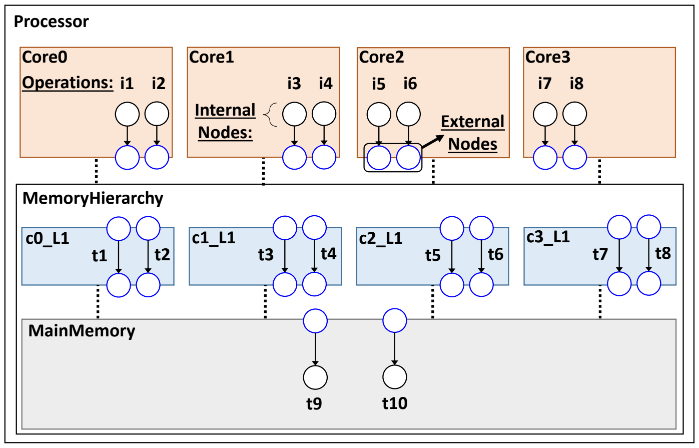


Figure 4.6: High-level graphical depiction of RealityCheck's model of Figure 4.1c's processor, showing examples of operations, internal nodes, and external nodes. `MemoryHierarchy`'s operations are not shown for brevity.

approach is to allow the teams working on each module to specify the axioms that actually hold in their modules, and then verify that the composition of the modules correctly maintains required MCM orderings. However, all axioms in PipeCheck's μspec specifications have omniscient visibility of the processor. There is no way for users to write axioms that are scoped to one portion of the design.

RealityCheck solves the above problems through its approach to modularity, hierarchy, and abstraction. The next section provides a high-level overview of how it does so.

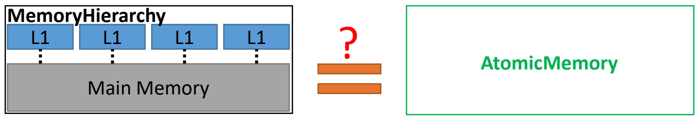


Figure 4.7: Interface verification consists of checking a set of implementation modules against an abstract interface to verify (up to a bounded number of operations) whether the implementation modules satisfy the interface specification. This figure depicts the interface verification of the `MemoryHierarchy` module and its constituent submodules against the `AtomicMemory` interface.

4.3 RealityCheck Overview

Figure 4.5 shows the high-level block diagram of RealityCheck. RealityCheck can be run in one of two ways: (i) for *litmus test verification* to verify a modular microarchitectural ordering specification against an ISA-level MCM specification for a specific litmus test, or (ii) for *interface verification* to verify the microarchitectural ordering specification of design components against the ordering specification of their abstract interfaces (Section 4.4). The latter use case enables a module to be verified independently of the rest of the design. The five steps in RealityCheck operation (**Microarchitecture Tree Generation**, **Operation Assignment**, **Formula Generation**, **Translation to Z3**, and **Graph Generation**) are common to both litmus test verification and interface verification. The difference between the two cases lies in which modules are checked and which operations they are checked on. Operations represent the instructions or instruction-like quantities that a module operates on. For example, the operations of a core module would be instructions, but the operations of a memory module would be memory transactions. Figure 4.6 provides a high-level graphical depiction of RealityCheck's basic terms (including operations) for Figure 4.1c's processor.

Two inputs that are provided to RealityCheck in both litmus test verification and interface verification are the implementation axiom files and the module definition files. These files are specified in the μspec++ language (Section 4.5) developed as part of RealityCheck. The μspec++ language is based on the μspec language [LSMB16] developed by PipeCheck, but adds support to the language for modularity, hierarchy, and abstraction, much like C++ does to C. The module definition files (Section 4.5.2) specify μspec++ modules in a manner similar to a C++ `.h` file. Meanwhile, each implementation axiom file (Section 4.5.1) specifies the events relevant to a given module as well as orderings on these events, in a manner similar to a C++ `.cpp` file. μspec++ prevents users from writing omniscient axioms that violate the visibility constraints of the structure of their design (Sections 4.5.1 and 4.5.2). It also accomplishes translation from ISA-level litmus tests into the operations of lower-level modules using connection axioms (Section 4.5.2) and appropriate operation assignment (Section 4.6.2).

If verifying a litmus test, the test is provided as input in the `.test` format from prior work [LSMB16]. Meanwhile, if running interface verification, RealityCheck takes in a list of implementation-interface pairs. Each pair specifies an implementation module to verify against an interface specification, and their corresponding node mappings (Section 4.5.3).

The final input to RealityCheck (which is always provided to the tool) is the bound up to which to conduct verification. Similar to most prior automated hardware MCM verification work [LPM14, MLPM15, LSMB16, TML+17, MLMP17] but unlike PipeProof (Chapter 5), RealityCheck conducts bounded verification; i.e., it explores all possible executions that use up to the specified number of operations (per module). Thus, RealityCheck is excellent for bug-finding, as I show in my case studies in Section 4.8.4.

4.4 Abstraction and its Benefits

In RealityCheck, *interfaces* can be used to separate the specification of a component's functional behaviour from the details of its implementation. For example, users may want to abstract the behaviour of their memory hierarchy as a single atomic memory, as shown in Figure 4.7.

The use of interfaces has several benefits. First, any implementation of the interface can be verified against the interface specification *independently* of the rest of the system. This gives users a method to verify the correctness of a design component without needing to link it to a top-level litmus test. For example, Figure 4.7 shows the verification of the `MemoryHierarchy` module and its submodules (instances of other modules that exist within `MemoryHierarchy`) against the `AtomicMemory` interface. Interface verification enables easy localisation of bugs to a given module based on whether it satisfies its interface. Second, the use of interfaces facilitates sharing of IP blocks between vendors. A vendor can internally verify their implementation against its interface for correctness, and then share their interface with other vendors without having to share their internal implementation specification. Third, interfaces enable scalable verification. Instead of verifying the entire design at once (Figure 4.1c), which will likely result in an SMT formula too large for solvers to handle, interfaces enable verification to be split into multiple steps. Specifically, the design is first verified using the (likely) smaller and simpler interface specification of the component (Figure 4.1d) rather than its implementation. Then, the component implementation is separately verified against the interface specification up to a user-provided bound (Figure 4.7). These two verification queries can be run in parallel, and will likely be smaller SMT formulae than verifying the design all at once. This process can be repeated further down the hierarchy. For instance, if the L1 caches in Figure 4.7 had interfaces, those interfaces could be used when conducting interface verification in Figure 4.7. The L1 implementation could then be separately verified against its interface specification. This splitting of verification queries using interfaces can be done again and again to split verification into smaller problems, thus allowing it to scale.

Finally, interfaces allow implementations to be switched in and out easily. For example, if the user wants to introduce a new memory hierarchy (say, one with an L2 cache) to a previously verified version of Figure 4.1d, then all they need to do to ensure

```
ModuleID "Core".

DefineEvent 0 "IF".
DefineEvent 1 "EX".
DefineEvent 2 "WB".
DefineEvent External 3 "MemReq".
DefineEvent External 4 "MemResp".

Axiom "PO_Fetch":
forall microop "i1",
forall microop "i2",
ProgramOrder i1 i2 =>
  AddEdge ((i1, IF), (i2, IF), "").

Axiom "Req_Resp_PO":
forall microop "i1",
forall microop "i2",
EdgeExists ((i1, WB), (i2, WB), "") /\
NodesExist [(i1, MemResp); (i2, MemReq)] =>
  AddEdge ((i1, MemResp), (i2, MemReq),"").
```

Figure 4.8: Part of `simpleProc`'s `Core` module's implementation axiom file.

design correctness is to verify the new memory hierarchy against the `AtomicMemory` interface, independently of the rest of the design.

4.5 μspec++ Modular Design Specifications

This section explains the μspec++ domain-specific language using a pedagogical microarchitecture `simpleProc`, comprised of 3-stage pipelines connected to a single main memory.

4.5.1 Implementation Axiom Files

Each implementation axiom file specifies the events relevant to a given module. Figure 4.8 shows part of the implementation axiom file for a module of type `Core`. The file begins with the module's type, and is followed by a list of the types of events that this module can observe and/or enforce orderings on, denoted using

`DefineEvent`. The rest of the file details the axioms which enforce various orderings on these events. Figure 4.8 shows two such axioms, `PO_Fetch` and `Req_Resp_PO`. These axioms are identical in syntax to the corresponding μspec axioms, but they only enforce orderings on the operations of the module in which they are declared. For example, if evaluating Figure 4.2's litmus test on a μspec++ design, an instance of the `Core` module representing Core 0 would only be able to see instructions `i1` and `i2` rather than all the instructions of the litmus test. In such a case, the `forall` quantifiers in the `PO_Fetch` axiom would evaluate to an AND over instructions `i1` and `i2`.

By default, events can only be viewed by the module instance in which they are declared, similar to private member variables in C++. For example, the `IF` and `WB` events in an instance of the `Core` cannot be seen outside that instance. If users write axioms in other modules that reference these events, RealityCheck will flag an error when parsing their specification, since these events are not visible outside the `Core` module. This capability allows designers to hide events internal to their design component from other modules in the system, just like in a real Verilog design. Since omniscient axioms (Section 4.2.2) by definition look at the internal events of other modules, they are illegal in such a setup. Thus, if users organise their microarchitectural ordering specifications into modules that reflect the structure of their design, RealityCheck will automatically prevent them from writing omniscient axioms that violate this design structure.

Meanwhile, a module's external events can be viewed by itself as well as by its parent module and any modules it may be connected to (see Section 4.5.2 for further details). Such events are labelled with the `External` keyword when they are declared in the implementation axiom file. For example, the `MemReq` and `MemResp` events in Figure 4.8's `Core` module are both external events. Thus, if the `Req_Resp_PO` axiom adds an edge between the `MemResp` and `MemReq` nodes of two instructions, that edge

```
Module Processor () {
  OperationType none
  Properties  { IsCore no }

  Submodules  {
    Core c0 (c : 0)
    Core c1 (c : 1)
    Core c2 (c : 2)
    Core c3 (c : 3)
    Mem mem ()
  }

  ConnectionAxioms {
    Axiom "instr_has_tran":
    forall microop "i" in "c0;c1;c2;c3",
    NodeExists (i, MemReq) =>
      exists transaction "j" in "mem",
        Mapped i j.
    ...
    Axiom "mapped_effects":
    forall microop "i" in "c0;c1;c2;c3",
    forall transaction "j" in "mem",
    Mapped i j =>
     (SameAddress i j /\ SameData i j /\
      (IsAnyRead i <=> IsAnyRead j) /\
      (IsAnyWrite i <=> IsAnyWrite j) /\
      SameNode (i, MemReq) (j, Req) /\
      SameNode (i, MemResp) (j, Resp)).
  }}
```

Figure 4.9: `simpleProc`'s `Processor` module definition.

will be visible outside the instance of the `Core` module in which the instructions reside. Figure 4.6 shows a graphical depiction of internal and external events/nodes.

4.5.2 Module Definition Files

Figure 4.9 shows the module definition of `simpleProc`'s top-level `Processor` module. A module definition file specifies the module's operation type, properties, submodules, and connection axioms.

```
Axiom "Mem_Writes_Path":
forall transaction "i",
    NodeExists (i, Req) /\ IsAnyWrite i =>
    AddEdges [((i, Req), (i, ViCL_C), "");
              ((i, ViCL_C), (i, ViCL_E), "");
              ((i, ViCL_C), (i, Resp), "")].
```

Figure 4.10: An implementation axiom of `simpleProc`'s `Mem`.

Operation Types and Properties

The operations in each module have some type, specified using the `OperationType` keyword. For example, `simpleProc`'s `Core` module has an operation type of `microop`, since it deals with instructions, while the `Mem` module has an operation type of `transaction`. Users can add additional operation types. The `Processor` module in Figure 4.9 has an operation type of `none`, a special identifier indicating that it has no operations. This is because the `Processor` module serves only to encapsulate the other modules in the system.

When quantifying over operations, RealityCheck verifies that the operation type used in the quantifier matches that of the operations over which the quantifier is being evaluated. So for instance, if the first `forall` quantifier in Figure 4.8's `PO_Fetch` axiom was replaced with `forall transaction "i1"`, RealityCheck would flag a type error.

A module's properties are certain fields that are shared across all instances of the module, similar to static variables of classes in C++. An example of a property is the `IsCore` property, which can be set to `yes` or `no`. RealityCheck uses this property in order to detect which modules are pipeline modules and thus should have litmus test instructions assigned to them when conducting litmus test evaluation.

Submodules

The submodules of a module are instances of other modules that exist within it. Submodules enable hierarchy in RealityCheck, allowing larger modules to be built

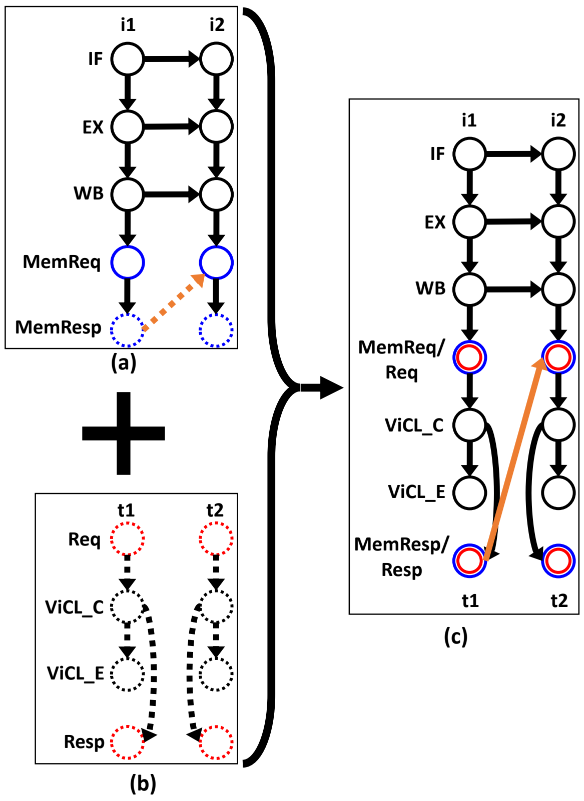


Figure 4.11: Effect of connection axioms in `simpleProc`, assuming 3-stage pipelines. (a) shows two stores in program order from a `Core`, while (b) shows two transactions from `Mem`. (c) shows the result when connection axioms merge the `MemReq` and `MemResp` nodes of `Core` with the `Req` and `Resp` nodes of `Mem`. Black nodes are internal nodes while red and blue nodes are external nodes. Dotted nodes and edges are not guaranteed to exist. Merged nodes are concentric circles.

using smaller ones. A module can evaluate μspec++ predicates on the operations of its submodules and observe their external events, but it cannot observe their internal events.

When instantiating a submodule, parameters may be passed to the instance to populate some instance-specific fields (similar to how parameters are passed to a constructor in C++). For example, when the `Processor` module instantiates `Core` modules as its submodules, it passes each of them an integer parameter `c` denoting their core ID. This parameter can then be used in the axioms of that module.

Connection Axioms

Submodules are connected to each other and to their parent module through connection axioms. The bottom of Figure 4.9 shows two example connection axioms. Connection axioms are similar to implementation axioms (Section 4.5.1), but have differences in their visibility. A module's connection axioms can observe the operations of the module itself and those of its submodules. They can observe all events of their module (both internal and external), but only the external events of any submodules.

Since connection axioms can observe the operations of multiple modules, their quantifiers must specify the domain over which they operate. Each quantifier provides a list of modules whose operations it applies to (`this` refers to the operations of the module itself). For example, the `forall` quantifier in the `instr_has_tran` axiom is evaluated on the operations from modules `c0`, `c1`, `c2`, and `c3`. Thus, the quantifier evaluates to an AND over all these operations, but does not apply to the operations in the `mem` module. Likewise, the `exists` quantifier in `instr_has_tran` applies only to the operations in the `mem` module.

Connection axioms are responsible for translating one module's operations to those of another, and for linking them together. For example, if a core is connected to memory, the core's instructions need to be translated and mapped to their corresponding memory transactions. Furthermore, there may be multiple possible translations for a given litmus test, e.g. a load may read from the store buffer in one execution (and thus generate no memory transactions), while in another execution it may read from memory, thus generating a memory transaction.

By checking all the different ways the connection axioms could be satisfied, RealityCheck examines all possible translations of operations between modules. For example, the `instr_has_tran` axiom in Figure 4.9 maps each instruction on the four cores which requests data from memory (signified by the existence of its external `MemReq` node) to some transaction in `mem`, denoted by the `Mapped` predicate. This reflects how in a real design, an instruction in the pipeline that accesses memory will generate a corresponding memory transaction. Other axioms not shown ensure that the mapping between instructions and memory transactions is 1-1. Mapping schemes other than 1-1 can be used where necessary; for instance, if a 64-bit load instruction is performed using two 32-bit read transactions.

If operations are mapped to each other 1-1, mapped pairs must agree on which addresses, values, etc are being accessed. They must also agree on the timing of their events. For example, the second connection axiom (`mapped_effects`) in Figure 4.9 enforces some of these orderings. It enforces that if an instruction is mapped to a memory transaction, then they must have the same address and read/write the same data value. It also enforces that load instructions map to read transactions and stores map to write transactions (through the `IsAnyRead` and `IsAnyWrite` predicates). In addition, it uses the `SameNode` predicate to link the `MemReq` event of the instruction `i` to the `Req` event of the transaction `j`. Likewise, the `MemResp` event of `i` is linked to the `Resp` event of `j`.

Linking two nodes with `SameNode` essentially merges the two nodes together, ensuring that they are exactly the same event. In this case, the processor's request to and response from memory are viewed from the memory side as an arriving request to which it sends a response. Further details about the semantics of `SameNode` are discussed in Section 4.6.4.

A Graphical Example

Figure 4.11 graphically depicts the effect of connection axioms in `simpleProc`. Black outlines denote internal nodes. External nodes are outlined in blue (in `Core`) or red (in `Mem`). In Figure 4.11a we have two stores `i1` and `i2` in program order from an instance of `Core`. Note that the `MemResp` nodes of `i1` and `i2` are dotted to indicate that these nodes are not guaranteed to exist. This reflects the fact that a `Core` cannot just assume that its memory requests will be responded to. The existence of the `MemResp` nodes must be enforced by axioms in the `Mem` module and communicated to the `Core` through connection axioms. The implementation axioms of the `Core` module in Figure 4.8 obey this convention; for instance, the `Req_Resp_PO` axiom does not add an edge between the `MemResp` event of `i1` and the `MemReq` event of `i2` unless the `MemResp` node of `i1` exists.

Meanwhile, in Figure 4.11b we have two memory transactions `t1` and `t2`, governed by the axioms of the `Mem` module – specifically, the axiom shown in Figure 4.10. This axiom enforces that if a write request is provided to the `Mem` module, then it is responded to. Note that all the nodes of `t1` and `t2` are dotted, indicating that none of them is guaranteed to exist. This reflects the fact that in `simpleProc`, memory will remain idle unless data is provided to or requested from it. Without connection axioms, no instructions must interact with memory, and so no nodes or edges in `Mem` are guaranteed to exist.

When the connection axioms in Figure 4.9 are enforced, the result is Figure 4.11c, where instruction `i1` is mapped to transaction `t1` and instruction `i2` is mapped to transaction `t2`. The `Req` node of transaction `t1` is now guaranteed to exist, because it is the same node as `i1`'s `MemReq` node (denoted by the concentric blue and red circles), which is guaranteed to exist by `Core`. Transaction `t2`'s `Req` node is similarly guaranteed to exist by `t2` being mapped to `i2`. Figure 4.10's axiom now enforces that both transactions are responded to, causing the `Resp` nodes of `t1` and `t2` to exist, and thus also causing the `MemResp` nodes of `i1` and `i2` to exist (since they are now merged with the `Resp` nodes). Finally, since the `MemResp` node of `i1` now exists, the `Req_Resp_PO` axiom of the `Core` module (Figure 4.8) now enforces (through the orange edge in Figure 4.11) that the `Core` must receive `i1`'s response from memory before sending `i2`'s request to memory.

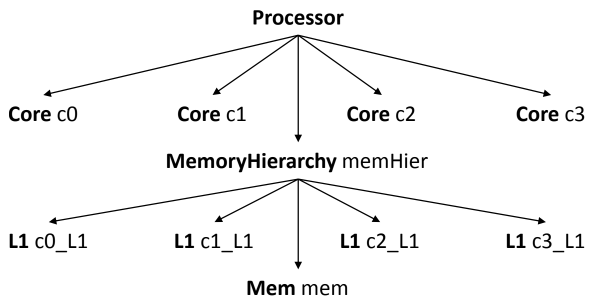


Figure 4.12: The microarchitecture tree of Figure 4.1c's processor.

4.5.3 Interface Specification and Node Mappings

Interfaces are specified in RealityCheck in a manner similar to other modules, but with some additional constraints. They are declared with the keyword `Interface` rather than `Module`. Interfaces cannot have submodules or connection axioms, as their goal is to provide a simple specification of component behaviour that does not delve into implementation details.

When verifying an implementation against an interface, the event types of the implementation must be mapped to those of the interface, so that the interface's properties can be checked on the implementation. Otherwise the interface and implementation would be referring to different events. For example, if verifying `MemoryHierarchy` against `AtomicMemory` as per Figure 4.7, one might map the request and response events of the memory hierarchy to the corresponding request and response events of the atomic memory. This list of node mappings must be provided along with an interface-implementation pair when it is input to RealityCheck for interface verification.

4.6 RealityCheck Operation

This section covers RealityCheck's five main steps (Figure 4.5).

4.6.1 Step 1: Microarchitecture Tree Generation

The first step in RealityCheck is **Microarchitecture Tree Generation**, which creates a tree of μspec++ module instances (i.e., copies) according to the module definition files and interface/implementation axiom files. Figure 4.12 shows the microarchitecture tree for the processor from Figure 4.1c. The root of the tree is a user-specified top-level module (in this case, the `Processor` module). The children of the top-level module are its submodules, which may have submodules of their own. To generate the microarchitecture tree for a design, RealityCheck first instantiates a copy of the top-level module, and then recursively instantiates each of its submodules.

4.6.2 Step 2: Operation Assignment

Once the microarchitecture tree is generated, **Operation Assignment** generates and assigns operations to each module. The design's various axioms are subsequently evaluated over these operations, with the visibility restrictions enforced by μspec++ detailed earlier.

RealityCheck assigns a number of operations to each module equal to the bound specified by the user as input. For instance, if assigning operations to Figure 4.1b's processor for a bound of 4, there would be 4 operations assigned to each of the 4 cores, each of the L1s, and to main memory, for a total of $16 + 16 + 4 = 36$ operations for the entire design. The bound specified as input to RealityCheck is the *maximum* number of operations per module that can exist in any verified execution, so an execution may use only some of the operations per module. RealityCheck accomplishes this by associating every operation with an implicit `IsNotNull` predicate, and enforcing that axioms only apply to non-null operations. This is the approach used by tools like Alloy [Jac12].

In litmus test verification, the design's `Core` modules (identified by the `IsCore` property) are assigned the litmus test instructions corresponding to their core. These litmus test instructions are *concrete*; their type, address, and value are dictated by the litmus test and cannot change. However, as Section 4.5.2 covers, verification of the litmus test must cover all possible translations (up to a bound) of these litmus test instructions to the operations of lower-level modules. RealityCheck accomplishes this translation by having operations in modules other than cores be *symbolic*, and having connection axioms enforce requirements on them based on the instructions they are (directly or indirectly) mapped to. Symbolic operations are abstract operations which can have any type (e.g., read, write, etc), address, or value, as long as the design's axioms are maintained. For example, if an operation from the `mem` module is mapped to an instruction from one of the `Core`s, the connection axioms (Figure 4.9) ensure that the symbolic operation in `mem` has the same type (read/write), address, and data as the instruction in the `Core`, effectively translating the `Core`'s instruction into a memory transaction.

Meanwhile, in interface verification, all operations of involved modules are symbolic. Thus, in that case, RealityCheck verifies that an implementation satisfies its interface for all possible combinations of operations up to the bound.

4.6.3 Step 3: Formula Generation

In Formula Generation, RealityCheck takes the conjunction of every module's implementation axioms and connection axioms, grounding quantifiers by translating `forall`s into ANDs and `exists` into ORs over each quantifier's domain, i.e., the operations being quantified over. RealityCheck conducts some preliminary simplification on the resultant representation, and then converts it (as described below) into an SMT formula checkable by the Z3 SMT solver [dMB08].

4.6.4 Steps 4 & 5: Translate to Z3 and Graph Generation

RealityCheck translates AND, OR, and NOT operators to their Z3 equivalents. Each predicate is mapped to a Z3 Boolean variable, except for `SameNode` (explained below). RealityCheck uses Z3's Linear Integer Arithmetic (LIA) theory to enforce happens-before orders. Each μhb node has two variables in Z3. The first is a Boolean variable dictating whether or not the node exists. The second is an integer variable denoting the timestamp of the node in the microarchitectural execution. An edge from a node `s` to a node `d` is translated to a constraint `e_s` < `e_d` where `e_s` and `e_d` are the integer variables denoting the timestamps of `s` and `d` respectively.

The `SameNode` predicate requires special handling, as it is not just a Boolean predicate, but also enforces that two nodes be merged together. If the user declares two nodes to be the same node in their μspec++, RealityCheck first creates a bi-implication between their Boolean variables to ensure that if one exists, so does the other (and vice versa). Then, RealityCheck adds a constraint that the integer variables denoting the timestamps of the nodes must have the same value. Together, these two constraints ensure that the two nodes in the `SameNode` predicate are treated as the same node.

If Z3 finds a satisfying assignment to the generated formula, the assignment represents an acyclic graph where nodes with their Boolean variables set to true exist, and where edges exist between nodes `s` and `d` if the integer variable for `s` is less than the integer variable for `d`. In such a case, RealityCheck parses the satisyfing assignment to generate a μhb graph that the user can view and use for debugging. If Z3 cannot find a satisfying assignment, then no acyclic μhb graph satisfying the constraints exists, and the outcome under consideration is unobservable (up to the user-specified bound).

4.7 RealityCheck Usage Flows

RealityCheck may be used to verify a component against its interface in isolation (Section 4.4), independent of the rest of the design. This capability of per-module verification enables RealityCheck to adapt to multiple design flows. If used at early-stage design time, users may first come up with a shallow design specification where all modules are represented by their interfaces, and then progressively replace modules with their submodules and implementations to create a more detailed design over time. Interface verification can be used to check implementations for correctness as the design becomes more detailed in this "outside-in" approach. If interface verification finds bugs, then additional axioms should be added to the implementation until interface verification succeeds. On the other hand, if an implementation (or RTL) already exists, users may favour an "inside-out" approach, where they first model one or a few modules deep in the system, and progressively add more modules and hierarchy to the specification until the entire design is modelled. In either case, some of the module boundaries in the processor may not be decided when engineers begin to write the specification, and may only become evident over time. RealityCheck's support for modularity, hierarchy, and abstraction ensures that this is not a problem. Engineers can easily add, remove, or replace RealityCheck modules or interfaces with different implementations or combinations of other modules as their design progresses, without needing to overhaul the entire specification each time.

When decomposing a module into submodules, users should try and minimise communication between the submodules. If such decomposition proves difficult, or would result in submodules with minimal internal functionality and large amounts of communication between them, it may be better to leave the module as a single unit, i.e. no submodules. The more internal events one can hide using abstraction, the faster verification will be (generally speaking). So for instance, RealityCheck will be fast for a microarchitecture that has a simple request-response interface to memory where all other memory functionality is hidden. Likewise, it will be slower for a speculative microarchitecture where coherence invalidations are propagated up to the pipeline. Nevertheless, RealityCheck is still capable of verifying such microarchitectures.

In the case where RTL exists prior to creating a RealityCheck model, users may use a tool like RTLCheck (Chapter 3) to check that the axioms they are writing for modules are sound for suites of litmus tests. RealityCheck's μspec++ axioms are a better fit for translation to SVA assertions using a tool like RTLCheck than the μspec axioms of PipeCheck are. PipeCheck's μspec axioms are omniscient (Section 4.2.2) and have no notion of an RTL implementation's structural modularity. Thus, SVA assertions generated from them must be evaluated over the entire RTL implementation. This greatly limits the scalability of these generated properties, as seen by the lack of complete proofs for some generated properties even for an implementation as small as Multi-V-scale (Section 3.9). Meanwhile, RealityCheck's μspec++ axioms are scoped to individual modules, so SVA assertions generated from such μspec++ axioms will be scoped to individual modules as well. As a result, these generated assertions will not need to be evaluated over the entire RTL implementation, but only over the modules to which the relevant μspec++ axiom is scoped. This will improve the verification performance of the generated SVA assertions, paving the way for scalable automated MCM verification of RTL in the future.[2]

[2]RTLCheck translates μspec axioms rather than μspec++ axioms because RTLCheck was developed chronologically before RealityCheck. In fact, the low scalability of RTLCheck's generated properties when they were run under the JasperGold verifier was a major impetus for the development of RealityCheck.

Table 4.1: Microarchitectures Evaluated Using RealityCheck

Name	**Pipelines**	**Mem. Hierarchy**	**MCM**
simpleProc	inOrderCore	unifiedMem	SC
cacheProc	inOrderCore	L1Hier	SC
simpleProcTSO	sbCore	unifiedMem	TSO
cacheProcTSO	sbCore	L1Hier	TSO
simpleProcRISCV	rvwmoCore	unifiedMem	RVWMO
cacheProcRISCV	rvwmoCore	L1Hier	RVWMO
heteroProcRISCV	2 sbCore-RISCV, 2 rvwmoCore	L1Hier	RVWMO

4.8 Methodology and Results

4.8.1 Methodology

RealityCheck is written primarily in Gallina, the functional programming language of Coq [Coq04]. It also includes some OCaml, specifically the code to translate μspec++ formulae into Z3 using the Z3 OCaml API. RealityCheck builds on PipeCheck's μspec parsing and axiom simplification [LSMB16], adding support for the various μspec++ features discussed in Section 4.5. In contrast to prior work [LSMB16] that used a basic SMT solver written in Gallina that supported μspec, RealityCheck utilises the state-of-the-art Z3 SMT solver [dMB08] to check its SMT formulae. RealityCheck's Gallina code is extracted to OCaml using Coq's built-in extraction functionality, and compiled along with RealityCheck's native OCaml code and Z3's OCaml API into a standalone binary that can call into Z3 to solve SMT formulae. Experiments were run on an Ubuntu 18.04 machine with 4 Intel Xeon Gold 6230 processors (80 total cores) and 1 TB of RAM. Each run of RealityCheck only utilises one core at time, but multiple instances of it can be run in parallel.

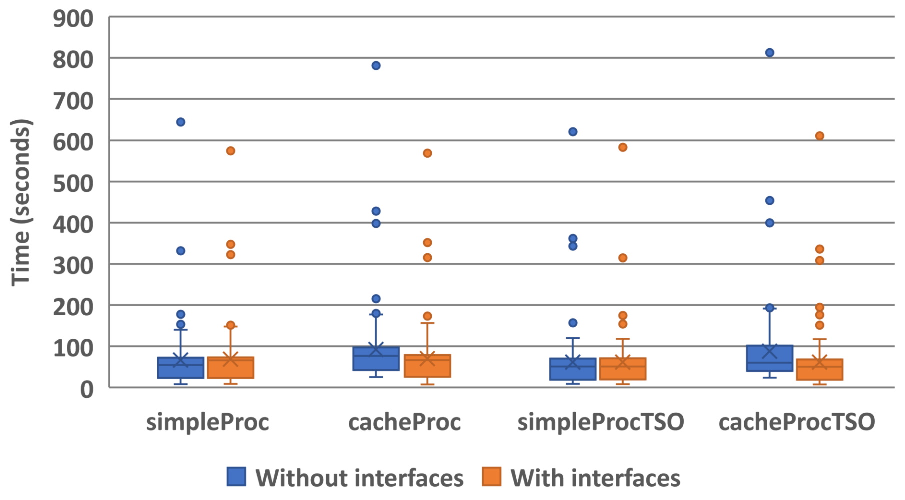


Figure 4.13: Verification times for 95 SC/TSO litmus tests across four microarchitectures implementing the SC and TSO MCMs. Each column represents the runtimes for the 95 tests for one particular configuration. The box represents the upper and lower quartile range of the data. Dots represent points lying beyond 1.5x the interquartile range (the extent of the whiskers) from the ends of the box. RealityCheck verifies 92 of the 95 litmus tests across all configurations in under 4 minutes each. The use of interfaces for abstraction reduces runtime by over 24% for the `cacheProc` and `cacheProcTSO` microarchitectures.

4.8.2 Verifying Litmus Tests

Table 4.1 lists the 7 microarchitectures (each with 4 cores) on which I ran RealityCheck. The first six microarchitectures are comprised of the possible combinations of three pipelines and two memory hierarchies. The three pipelines are: (i) `inOrderCore`, an in-order 5-stage pipeline that performs memory operations in program order, (ii) `sbCore`, an in-order 5-stage pipeline with a store buffer from which it can forward values, and (iii) `rvwmoCore`, an out-of-order RISC-V pipeline which implements RISC-V's RVWMO weak memory model [RIS19]. `sbCore` is capable of reordering writes with subsequent reads, as allowed by TSO (Total Store Order), the consistency model of Intel [Int13] and AMD x86 processors. `rvwmoCore` is more relaxed; it allows any loads and stores to different addresses to be reordered (in the absence of fences), while preserving address, data, and control dependencies as required by RVWMO.

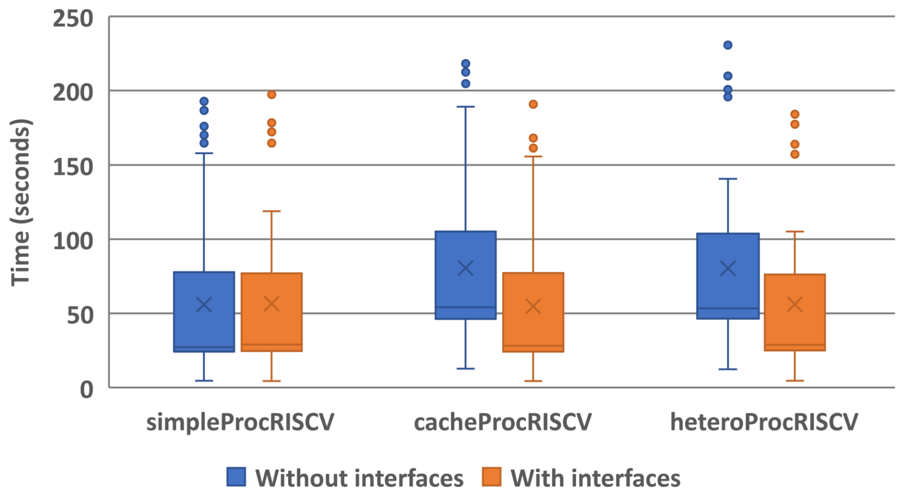


Figure 4.14: Verification times for 98 RISC-V litmus tests across three microarchitectures implementing the RISC-V RVWMO MCM. RealityCheck verifies each litmus test in under 4 minutes. The use of interfaces for abstraction reduces runtime by 30% or more for the `cacheProcRISCV` and `heteroProcRISCV` microarchitectures.

`rvwmoCore` supports coalescing of stores in its store buffer, and also supports all 8 of the RISC-V base ISA's fences for ordering memory loads and stores. The two memory hierarchies I model are `unifiedMem`, a single unified memory, and `L1Hier`, which consists of an L1 cache module backed by a module for a unified memory. The L1 cache in `L1Hier` is a non-blocking cache. It models requests for data from main memory, cache occupancy, and coalescing of stores before writing back to memory.

The final microarchitecture I evaluate RealityCheck on is `heteroProcRISCV`, a heterogeneous RISC-V processor. `heteroProcRISCV` has 2 out-of-order `rvwmoCore` pipelines and 2 in-order `sbCore` pipelines modified for the RISC-V ISA. To modify `sbCore` for RISC-V, I changed it to only enforce RISC-V fences that order writes with respect to subsequent reads, and to treat all other fences as nops. This is because the orderings enforced by other RISC-V fence types are already maintained by `sbCore` by default.

I also created four interfaces: one for each pipeline (`inOrderInt`, `sbInt`, and `rvwmoInt`) and one for the `L1Hier` memory hierarchy (`AtomicMemory`). Pipeline

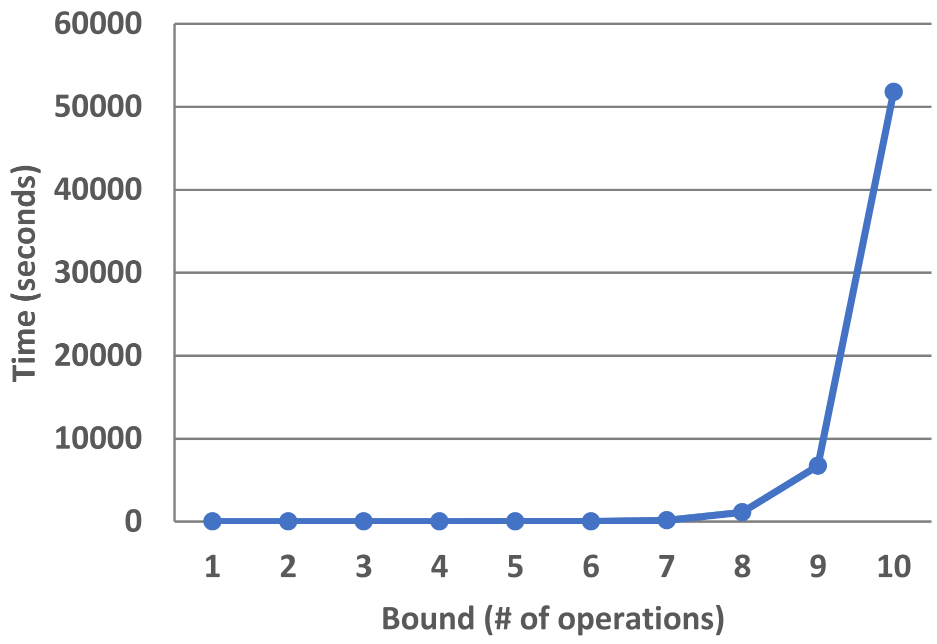


Figure 4.15: Runtimes for verifying `L1Hier` against `AtomicMemory` with varying bounds. Low bounds (e.g. 3-4) are sufficient to find common bugs.

interfaces like `inOrderInt` reduce each pipeline module to its requests to and responses from memory (and its dependency orderings, in the case of `rvwmoInt`). Meanwhile, `AtomicMemory` abstracts `L1Hier` as a unified memory. These interfaces help verification scale as discussed below.

SC and TSO Results

Figure 4.13 shows RealityCheck's runtimes (as a box-and-whisker plot) for a suite of 95 SC/TSO litmus tests on the four SC and TSO microarchitectures, both with and without interfaces. These results use a bound of up to 11 operations per module. The litmus tests in the SC/TSO suite come from a variety of sources, including existing x86-TSO suites [OSS09] and automatically generated tests from the diy tool [diy12]. As Figure 4.13 shows, RealityCheck's verification of litmus tests is quite fast, despite the increased detail of its microarchitectural specifications when compared to PipeCheck. 92 of the 95 tests are verified by all 8 configurations in under 4 minutes each. The three remaining tests (`co-iriw`, `n3`, and `iwp27`) take longer as they have a large number of instructions (e.g. `n3` has 9 instructions) and/or possibilities to consider. However, RealityCheck still verifies them under all configurations in less than 14 minutes each.

Figure 4.13 also shows how the use of interfaces provides significant reductions in overall litmus test verification runtime. Pipeline interfaces and `AtomicMemory` can be used to abstract away portions of each design for litmus test verification. The use of abstraction reduces the total time to verify all litmus tests by 24.2% for `cacheProc`, 0.6% for `simpleProcTSO`, and 29.7% for `cacheProcTSO`. Meanwhile, the use of abstraction increases runtime for `simpleProc` by 3.7%, illustrating that interfaces may not reduce verification time for very simple designs. The runtime savings are much higher for `cacheProc` and `cacheProcTSO` because they use the `AtomicMemory` interface to abstract `L1Hier`. `L1Hier` is relatively detailed when compared to `AtomicMemory`, so verification using `AtomicMemory` takes much less time.

RVWMO Results

Figure 4.14 shows RealityCheck's runtimes for 98 RVWMO litmus tests on my 3 RVWMO microarchitectures, both with and without interfaces. These results use a bound of up to 11 operations per module. The RVWMO litmus tests used are generated using an automated litmus test synthesis tool [LWPG17]. Given a formal MCM specification, the tool generates all litmus tests (up to a bound) for that MCM that satisfy a *minimality criterion*: no synchronization mechanism in the test can be weakened without causing new behaviours to become observable. As a result, the generated tests provide excellent coverage of MCM corner cases. The litmus tests I use for RVWMO are the set of litmus tests up to 6 instructions long generated by this tool for the RVWMO MCM.

As Figure 4.14 shows, RealityCheck verifies each litmus test in all 6 configurations in under 4 minutes per test. The maximum time per test is lower under RVWMO than SC or TSO because my largest RVWMO litmus test is 6 instructions long

Table 4.2: Interface verification times for pipeline modules (bound of 15)

inOrderCore	**tsoCore**	**riscvCore**
< 1 sec.	18 sec.	42 minutes

(compared to e.g., the `n3` TSO litmus test which has 9 instructions). Similar to the SC and TSO microarchitectures, interfaces reduce verification time by 32.1% for `cacheProcRISCV` and 30.0% for `heteroProcRISCV`, while increasing runtime by 1.0% for `simpleProcRISCV`.

The use of interfaces for abstraction depends on the implementations of those interfaces being verified against the interface specifications. I present results on those next.

4.8.3 Interface Verification

I conducted interface verification of three pipelines (`inOrderCore`, `sbCore`, and `rvwmoCore`) against their respective interfaces. Table 4.2 shows interface verification times for these pipelines (bound of 15). Interface verification of `rvwmoCore` takes longer than interface verification for the other two pipelines because the RISC-V pipeline is substantially more complicated. Nevertheless, its interface verification completes in 42 minutes.

I also verified the `L1Hier` memory hierarchy against `AtomicMemory`. Figure 4.15 shows interface verification runtime for `L1Hier` against `AtomicMemory` with varying bounds. `MemHier` interface verification runtimes at higher bounds are significantly larger than litmus test runtimes. For instance, interface verification of `L1Hier` at a bound of 10 takes over 14 hours. This is not surprising, as interface verification checks the implementation for all possible combinations of operations up to the user-specified bound. So for example, if verifying `L1Hier` against `AtomicMemory` with a bound of 10, one is checking all possible combinations of up to 10 transactions. This is essentially verifying all possible "programs" (from the perspective of memory) of up to

10 operations, which is far more than the possible memory transaction combinations that could result from a single litmus test.

The high runtimes for interface verification of `MemHier` are not as big an issue as they may initially seem. As Section 4.8.4 below shows, bugs in implementations which cause them to not match their interfaces are detectable at lower bounds, and are found quickly even at higher bounds. Thus, even if interface verification has not terminated, if it does not find a bug quickly, the design is likely to be correct. Furthermore, interface verification can be run in parallel with both litmus test verification and with interface verification of other modules, making it well placed to take advantage of large compute clusters. Finally, interface verification of a module only needs to be run once, not once per litmus test. As the number of litmus tests run increases, the time saved from using interfaces for abstraction will draw closer to the time for interface verification.

4.8.4 Bug Finding

To test how quickly interface verification can find bugs, I performed three case studies where I added a bug to the implementation of a component and then verified it against its interface. The first bug I added was to remove the axiom in `L1Hier`'s L1 cache which ensured that it could only have one value for a given address at any time. RealityCheck discovered this bug when verifying `L1Hier` against `AtomicMemory` at a bound of 3 in less than a second. Even when the bound was increased to 15, RealityCheck still found the bug in under 2 minutes. The second bug I added was to remove the axiom in `L1Hier`'s L1 cache which prevented it from dropping dirty values without writing them back. RealityCheck discovered this bug during interface verification at a bound of 4 in less than a second. Once again, even when the bound was increased to 15 operations, RealityCheck still caught the bug in less than 2 minutes. The final bug I added was to try and verify `sbCore` against `inOrderInt`. This should fail because `inOrderInt` requires program order to be preserved, while `sbCore` relaxes write-read ordering. RealityCheck duly discovered the bug at a bound of 15 in under a second.

4.9 Chapter Summary

Modern processors are complex parallel systems which incorporate components built by many different teams (and possibly multiple vendors), and they require stringent MCM verification to ensure their correctness. In prior work, PipeCheck developed a methodology for automated microarchitectural MCM verification, but its monolithic approach is ill-suited to the verification of large designs (like those of commercial processors) for three reasons. Firstly, PipeCheck's monolithic verification does not scale to detailed models of large designs due to the NP-completeness of SMT solvers. Secondly, PipeCheck's specifications are also monolithic, with no capability to explicitly denote component boundaries. This is problematic because processors are usually designed by a combination of distinct teams, where each team is responsible for one or a few design components. Each team will know the details of its own component to write a specification for it, but they may know very little about the components designed by other teams. As a result, it is very hard to write a monolithic specification for the entire processor, as no one individual has detailed knowledge of the entire design. Finally, PipeCheck's monolithic specifications encourage the use of axioms that exercise a omniscient view of the design. These axioms often reflect architectural intent rather than the reality of what the design does, and can lead to unsound verification if the architectural intent does not match reality. All these problems can be solved through the development of an automated microarchitectural MCM verification approach that incorporates modularity, hierarchy, and abstraction.

In response, this chapter presents RealityCheck, the first methodology and tool for automated formal MCM verification of modular hardware design specifications. The key idea behind RealityCheck is to leverage the structural modularity and hierarchy present in today's hardware designs to provide scalable automated MCM verification (up to a bound) of microarchitectural orderings. RealityCheck provides support in its novel specification language μspec++ for modularity, hierarchy, and abstraction. Users of RealityCheck can encapsulate orderings enforced by a component into modules which can be composed with and instantiated inside each other to create larger modules. This makes RealityCheck well-suited to the distributed nature of processor design. It also helps prevent users from writing omniscient axioms that do not reflect what the design is actually doing. Furthermore, RealityCheck enables each component to be verified against its interface specification independently of the rest of the system, and regardless of whether the rest of the design specification exists. This enables verification of the entire design to be split into multiple smaller verification problems, allowing verification to scale. As the results in this chapter demonstrate, RealityCheck is capable of automatically verifying hardware designs across a range of litmus tests, and it can detect bugs extremely quickly. Finally, RealityCheck fulfills the requirement for detailed design verification in a progressive verification flow (Chapter 6) for MCM properties in parallel processors.

Chapter 5

Automated All-Program MCM Verification[1]

That one may smile, and smile, and be a villain;
—William Shakespeare
Hamlet

Chapters 3 and 4 cover how to verify the soundness of microarchitectural ordering models and how to conduct scalable microarchitectural MCM verification respectively. Verification of microarchitectural model soundness and scalable verification procedures are both critical for the MCM verification of real-world processors. Equally important, however, is the issue of MCM verification coverage addressed by this chapter.

A hardware design must obey its MCM for any program that it may run. Thus, to ensure that parallel programs run correctly, verification of hardware MCM implementations would ideally be *complete*; i.e., verified as being correct across all possible executions of all possible programs. However, no prior automated approach was capable of such complete verification. For instance, PipeCheck [LPM14, LSMB16] (Section 2.4) can only verify one litmus test at a time. Since there is an infinite

[1]An earlier version of the work in this chapter was previously published and presented by myself at the MICRO-51 conference [MLMG18]. I was the first author on the publication.

number of possible programs, PipeCheck is incapable of proving correctness across all programs.

To help fill this verification gap, this chapter presents PipeProof, a methodology and tool for complete MCM verification of an axiomatic microarchitectural ordering specification against an axiomatic ISA-level MCM specification. PipeProof can automatically prove a microarchitecture correct across all programs, or return an indication (often a counterexample) that the microarchitecture could not be verified. To accomplish unbounded verification, I developed the novel *Transitive Chain Abstraction* to represent microarchitectural executions of an arbitrary number of instructions using only a small, finite number of instructions. With the help of this abstraction, PipeProof proves microarchitectural correctness using an approach based on Counterexample-Guided Abstraction Refinement (CEGAR) [CGJ$^+$00]. In addition, to my knowledge, PipeProof is the first ever automated refinement checking methodology and tool for pairs of axiomatic models of executions. PipeProof's implementation includes algorithmic optimizations which improve runtime by greatly reducing the number of cases considered. As a proof-of-concept study, this chapter includes results for modelling and proving correct simple microarchitectures implementing the SC and TSO MCMs. PipeProof verifies both case studies in under an hour.

5.1 Introduction

A key challenge in parallel hardware design is ensuring that a given microarchitecture (hardware design) obeys its ISA-level MCM. If the hardware does not respect its MCM in all possible cases, then the correct operation of parallel programs on that implementation cannot be guaranteed.

The ramifications of MCM bugs in parallel processors necessitate verifying that the hardware correctly implements its MCM. Such verification would ideally be *complete*;

i.e., it would cover all possible executions of all possible programs. However, truly complete verification is extremely difficult. The only prior all-program consistency proofs of hardware are implemented in interactive proof assistants [VCAD15, CVS+17] (Section 2.3.4), which require significant manual effort.

Automated approaches make it much easier for microarchitects to verify their designs. However, no prior automated approach was capable of all-program MCM verification. Dynamic verification approaches [HVML04, MS09] (Section 2.3.5) only examine a subset of the possible executions of any program they test, so they are incomplete even for those programs. Formal MCM verification approaches look at all possible executions of the programs they check, but all such approaches that are automated have only been able to conduct verification of implementations for a *bounded* set of programs. The verified programs in such an approach may be a suite of litmus tests [AMT14, LPM14, MLPM15, LSMB16, TML+17, MLMP17], or all programs smaller than a certain bound (∼10 instructions) [WBSC17].

Critically, litmus test-based or bounded verification only ensures that the implementation will correctly enforce orderings for the verified programs themselves. It does *not* ensure that the implementation will correctly enforce required orderings for all possible programs. These incomplete approaches have missed important bugs because the set of programs they verified did not exercise those bugs [WBSC17]. This necessitates an automated approach capable of conducting *all-program* MCM verification of microarchitectural implementations.

To fill this need for all-program microarchitectural MCM verification, this chapter presents PipeProof[2], a methodology and automated tool for unbounded verification of axiomatic microarchitectural ordering specifications [LSMB16] against axiomatic ISA-level MCM specifications [AMT14]. PipeProof uses an approach based on Satisfiability Modulo Theories (SMT) [BSST09] and Counterexample-Guided Abstraction

[2]PipeProof is open-source and publicly available at github.com/ymanerka/pipeproof.

Refinement (CEGAR) [CGJ$^+$00]. PipeProof's unbounded verification covers all possible programs, core counts, addresses, and values. The key to PipeProof's unbounded verification is its novel *Transitive Chain Abstraction*, which allows PipeProof to inductively model and verify the infinite set of program executions that must be verified for a given microarchitecture. As its output, PipeProof either provides a guarantee that the microarchitecture is correct, or returns an indication that the microarchitecture could not be verified. In addition, to my knowledge, PipeProof is the first ever automated refinement checking methodology and tool for pairs of axiomatic models of executions. As such, PipeProof makes advances in refinement checking in addition to its contributions to MCM verification.

The rest of this chapter is organised as follows. Section 5.2 covers PipeProof's procedure for proving microarchitectural correctness. Section 5.3 presents the supporting proofs and modelling techniques leveraged by the microarchitectural correctness proof. Section 5.4 covers optimizations that reduce PipeProof's runtime, while Section 5.5 discusses methodology and results. Section 5.6 discusses related work, and Section 5.7 summarises this chapter.

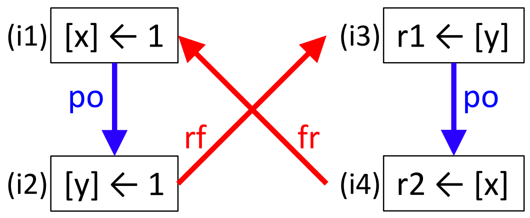
5.2 PipeProof Operation

PipeProof's goal is to prove that a microarchitectural ordering specification satisfies its ISA-level MCM specification across all executions of all programs. Section 2.3.1 provides background on the ISA-level MCM specifications used by PipeProof, while Section 2.4.2 provides background on the microarchitectural specifications PipeProof uses. A very brief refresher is provided below.

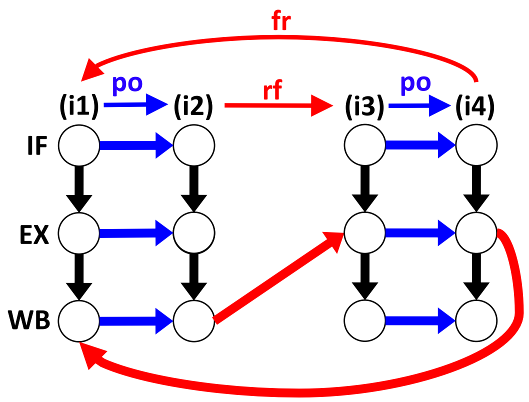
In PipeProof, ISA-level and microarchitectural executions are formally defined as follows:

Core 0	Core 1
(i1) [x] ← 1	(i3) r1 ← [y]
(i2) [y] ← 1	(i4) r2 ← [x]
SC forbids r1=1, r2=0	

(a) Code for litmus test `mp`



(b) ISA-level execution of `mp` forbidden under SC due to the cycle in the *po*, *rf*, and *fr* relations.



(c) μhb graph for `mp`'s forbidden outcome on the `simpleSC` microarchitecture. The corresponding ISA-level cycle is shown above the μhb graph.

Figure 5.1: Example litmus test, ISA-level execution, and μhb graph for the `mp` litmus test.

Definition 1 (ISA-Level Execution)**.** An ISA-level execution $(Instrs, Rels)$ is a graph. Nodes $Instrs$ are instructions, and edges $Rels$ between nodes represent ISA-level MCM attributes.

Definition 2 (Microarchitectural Execution)**.** A microarchitectural execution is a μhb graph $(Instrs, N, E)$. Nodes N represent individual sub-events in the execution of instructions $Instrs$. Edges E represent happens-before relationships between nodes.

ISA-level MCM specifications (Section 2.3.1) used by PipeProof are axiomatically defined in terms of the irreflexivity, acyclicity, or emptiness of certain relational patterns. For example, SC can be defined in this framework as $acyclic(po \cup co \cup rf \cup fr)$. Meanwhile, the microarchitectural ordering specifications used by PipeProof are sets of μspec axioms (Section 2.4.2). Figure 5.1 shows both the ISA-level execution (Figure 5.1b) and a μhb graph (Figure 5.1c) that correspond to the non-SC outcome of the `mp` litmus test (Figure 5.1a). The ISA-level execution has a cycle in the *po*,

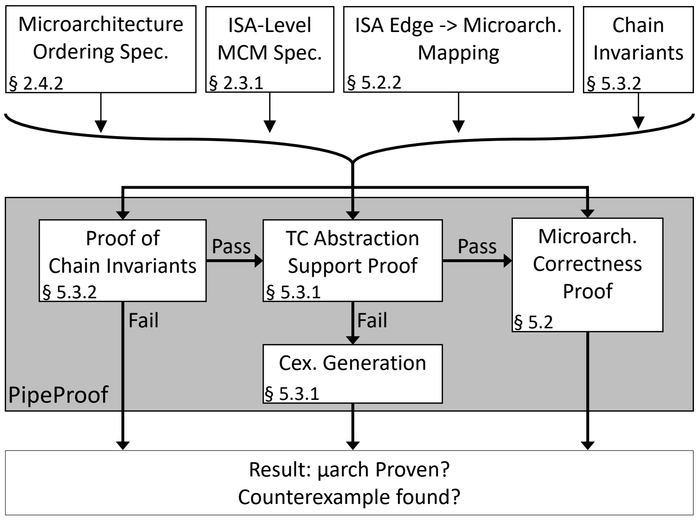

Microarchitecture Ordering Spec.
§ 2.4.2
ISA-Level MCM Spec.
§ 2.3.1
ISA Edge -> Microarch. Mapping
§ 5.2.2
Chain Invariants
§ 5.3.2
Proof of Chain Invariants
§ 5.3.2
Pass
TC Abstraction Support Proof
§ 5.3.1
Pass
Microarch. Correctness Proof
§ 5.2
Fail
Fail
Cex. Generation
§ 5.3.1
PipeProof
Result: µarch Proven? Counterexample found?

Figure 5.2: High-level block diagram of PipeProof operation. Components are annotated with the sections in which they are explained.

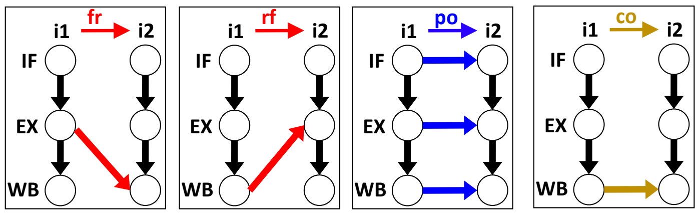


Figure 5.3: ISA-level relations can be analysed in terms of how a given microarchitecture enforces them. These four μhb graphs show the μhb edges between instructions that are enforced (directly or indirectly) by the mappings of ISA-level edges to the `simpleSC` microarchitecture.

rf, and fr relations, and so is forbidden by SC as one would expect. The μhb graph represents an execution of `mp` on a simple microarchitecture (henceforth referred to as `simpleSC`) with 3-stage in-order pipelines. The 3 stages in this pipeline are `Fetch` (`IF`), `Execute` (`EX`), and `Writeback` (`WB`). The corresponding ISA-level cycle (the same one as Figure 5.1b) is shown on top of the μhb graph. In `simpleSC`, loads access memory in the `Execute` stage and stores go to memory during `Writeback`. The μhb graph in Figure 5.1c contains a cycle, so the execution represented by it is unobservable on the microarchitecture being modelled, as expected of an SC microarchitecture.

```
Axiom "Mappings_po":
forall microop "i", forall microop "j",
HasDependency po i j => AddEdge ((i, Fetch), (j, Fetch), "po_arch").
```

Figure 5.4: Example mapping axiom for *po* ISA-level edges on `simpleSC`.

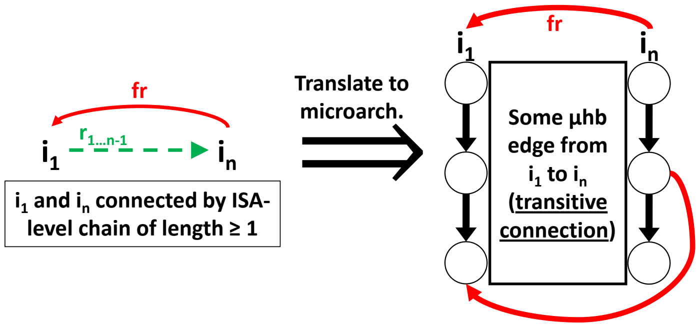

fr
r1…n-1
i1
in
i1 and in connected by ISA-level chain of length ≥ 1
Translate to microarch.
fr
i1
in
Some μhb edge from i1 to in (transitive connection)

Figure 5.5: A graphical example of the Transitive Chain (TC) Abstraction: all possible ISA-level chains connecting i_1 to i_n (left) can be abstracted as some μhb edge (the *transitive connection*) between the nodes of instructions i_1 and i_n (right). The red μhb edge is the microarchitectural mapping of the *fr* edge from i_n to i_1.

5.2.1 PipeProof Overview

Figure 5.2 shows PipeProof's high-level block diagram. The inputs to PipeProof are a set of μspec axioms describing microarchitectural orderings, an ISA-level MCM specification, mappings (to link ISA-level and microarchitectural executions), and chain invariants (to abstractly represent repeating ISA-level patterns). As its output, PipeProof will either prove the microarchitecture correct for all possible programs or return an indication that the microarchitecture could not be verified.

PipeProof's overall operation has three high-level steps:

1. Prove chain invariants correct (Section 5.3.2).

Then for each forbidden ISA-level pattern in the ISA-level MCM specification:

2. Prove Transitive Chain (TC) Abstraction support for the microarchitecture and the ISA-level pattern (Section 5.3.1).

3. Prove Microarchitectural Correctness for the microarchitecture and the ISA-level pattern (covered in the remainder of this section).

The proofs of TC Abstraction support and chain invariants are supporting proofs on which PipeProof's main *Microarchitectural Correctness Proof* builds. The Microarchitectural Correctness Proof proves Theorem 2 below.

Theorem 2 (Microarchitectural Correctness)**.** For each ISA-level execution $ISAExec := (Instrs, Rels)$ where $Rels$ exhibits a pattern forbidden by the ISA-level MCM, all microarchitectural executions $(Instrs, N, E)$ corresponding to $ISAExec$ are unobservable (i.e., their μhb graphs are cyclic).

The rest of this section describes the Microarchitectural Correctness Proof in detail, beginning with the structure of the ISA-level executions PipeProof verifies (Section 5.2.2) and their translation to equivalent microarchitectural executions (Section 5.2.3). The Microarchitectural Correctness proof uses an abstraction refinement approach that leverages the TC Abstraction (Section 5.2.4) to model executions. PipeProof's refinement process involves examining abstract counterexamples (Section 5.2.5) and refining the abstraction through concretization and decomposition (Section 5.2.6). The section concludes by explaining when PipeProof's algorithm is able to terminate (Section 5.2.7).

5.2.2 Symbolic ISA-Level Executions

PipeProof works with ISA-level executions that are similar to the ISA-level execution of `mp` in Figure 5.1b, but it uses *symbolic* instructions. In other words, the instructions in such ISA-level executions do not have specific addresses or values. The symbolic version of the ISA-level execution in Figure 5.1b would consist of four instructions $i1$, $i2$, $i3$, and $i4$, connected by the *po*, *rf*, and *fr* edges as shown, but nothing more would be known about the four instructions beyond the constraints enforced by the ISA-level relations. For instance, the instructions connected by *po* would be known to be on the same core, and the *rf* edge between $i2$ and $i3$ would enforce that $i2$ and $i3$

had the same address and value. However, the specific address and value of $i2$ and $i3$ could be anything. Such a symbolic ISA-level execution represents not only the ISA-level execution of `mp` in Figure 5.1b, but *any* ISA-level execution comprised of the cycle[3] $po; rf; po; fr$. Thus, verifying such a symbolic ISA-level execution checks the instance of the ISA-level pattern in that execution across all possible addresses and values, as required for complete verification.

5.2.3 Mapping ISA-level Executions to Microarchitecture

To verify that a forbidden ISA-level execution is microarchitecturally unobservable, one needs to translate the ISA-level execution to its corresponding microarchitectural executions.[4] PipeProof's complete verification requires such translation for any arbitrary ISA-level execution, not just a particular program instance. PipeProof's microarchitectural executions are similar to the μhb graph in Figure 5.1c, but like PipeProof's ISA-level executions, they operate on symbolic instructions which do not have specific addresses and values.

An ISA-level execution's instructions can be translated by instantiating the μspec microarchitectural axioms for those instructions. Translating ISA-level relations to microarchitecture is harder because the microarchitectural constraints implied by an ISA-level relation differ between microarchitectures. Thus, PipeProof requires user-provided *mappings* to translate individual ISA-level edges to their corresponding microarchitectural constraints. These mappings are additional μspec axioms that restrict the executions examined by PipeProof's solver to the microarchitectural executions where the mapped ISA-level edge exists between its source and destination instructions.

[3]A semicolon (;) denotes relational composition. For example, $r_1; r_2$ denotes a sequence of two ISA-level edges r_1 and r_2 where the destination instruction of r_1 is the source instruction of r_2.

[4]There are usually multiple microarchitectural executions corresponding to a single ISA-level execution.

Figure 5.3 shows the μhb edges enforced by mappings of the ISA-level edges fr, rf, po, and co on `simpleSC`. Figure 5.4 shows the mapping axiom for po edges on `simpleSC`, which translates a po edge between instructions i and j to a μhb edge between the `Fetch` stages of those instructions. Such an edge can be seen between $i1$ and $i2$ in the po case of Figure 5.3. This edge between the `Fetch` stages indirectly induces edges between the `Execute` and `Writeback` stages of the instructions as well, through other axioms from `simpleSC`'s μspec. These μhb edges are also shown in Figure 5.3, and reflect the in-order nature of `simpleSC`'s pipeline.

5.2.4 The TC Abstraction: Representing Infinite ISA-level Chains

Symbolic analysis and the use of mappings for ISA-level edges allow a single ISA-level cycle to be translated to the microarchitectural level for all possible addresses and values. However, there are an infinite number of possible ISA-level cycles that match a forbidden ISA-level pattern like $cyclic(po \cup co \cup rf \cup fr)$ (which are the executions forbidden by SC). Conceptually, all of these ISA-level cycles need to be verified as being unobservable in order to ensure that the microarchitecture is correct across all possible programs. Such unbounded verification is made possible by using inductive approaches. PipeProof achieves unbounded verification by inductively modelling executions using a novel *Transitive Chain (TC) Abstraction.*[5] Specifically, PipeProof uses the TC Abstraction to efficiently represent ISA-level chains (defined below):

Definition 3. An ISA-level *chain* is an acyclic sequence of ISA-level relations $r_1; r_2; r_3 ...; r_n$. An example ISA-level chain is $po; rf; po$ from Figure 5.1b.

[5]This abstraction is conceptually similar but completely distinct from the module-based abstraction covered in Chapter 4. In Chapter 4, my work uses interfaces to abstract away portions of the design specification to enable scalable verification. In this chapter, my work uses the Transitive Chain Abstraction to abstractly represent portions of microarchitectural executions, enabling all-program verification.

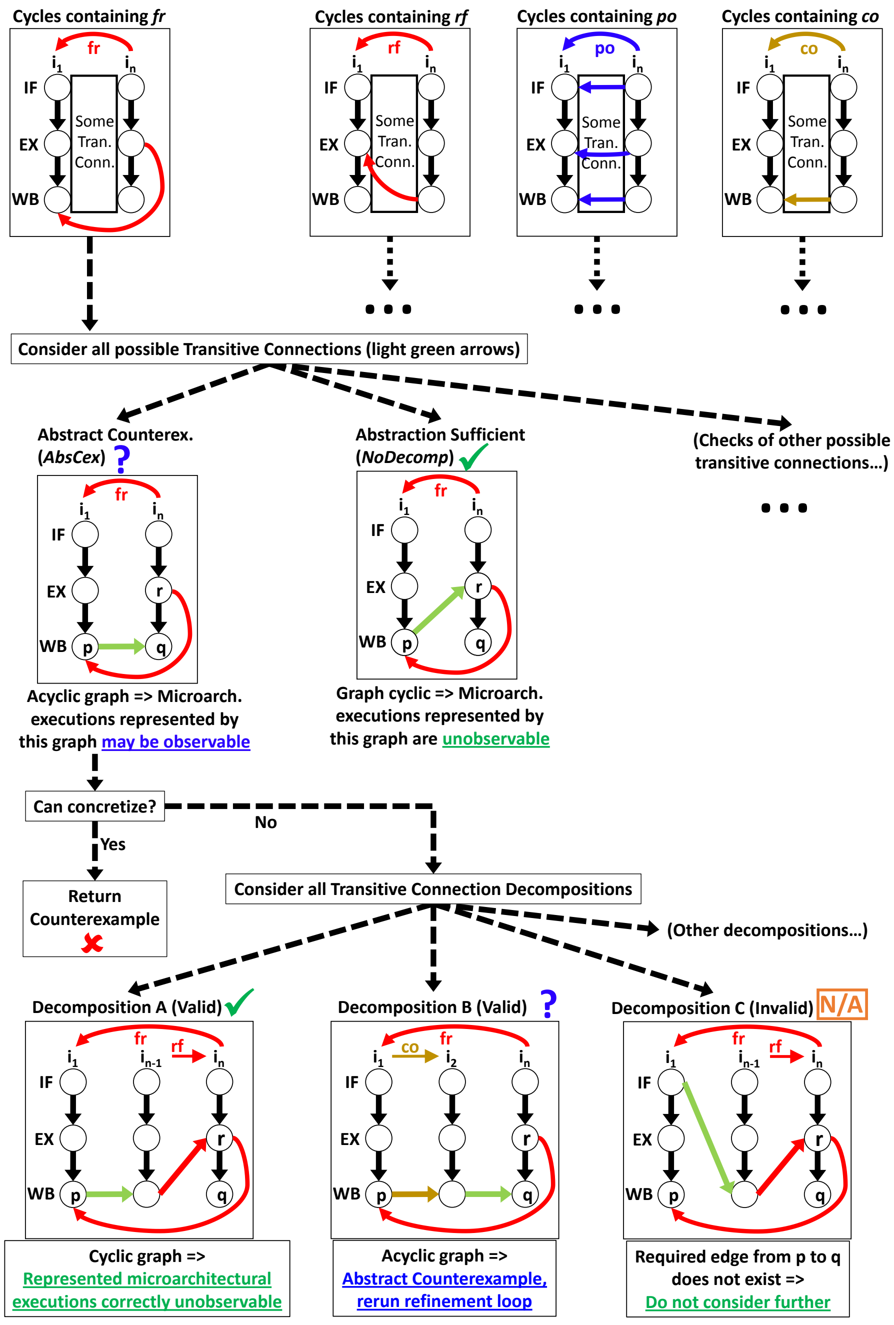

Cycles containing *fr*
Cycles containing *rf*
Cycles containing *po*
Cycles containing *co*
fr
rf
po
co
IF
EX
WB
Some Tran. Conn.
Consider all possible Transitive Connections (light green arrows)
Abstract Counterex. (*AbsCex*)
Abstraction Sufficient (*NoDecomp*)
(Checks of other possible transitive connections…)
Acyclic graph => Microarch. executions represented by this graph may be observable
Graph cyclic => Microarch. executions represented by this graph are unobservable
Can concretize?
Yes
No
Return Counterexample
Consider all Transitive Connection Decompositions
(Other decompositions…)
Decomposition A (Valid)
Decomposition B (Valid)
Decomposition C (Invalid)
N/A
Cyclic graph => Represented microarchitectural executions correctly unobservable
Acyclic graph => Abstract Counterexample, rerun refinement loop
Required edge from p to q does not exist => Do not consider further

Figure 5.6: PipeProof checking that non-unary ISA-level cycles forbidden by SC are unobservable on `simpleSC` with the help of the Transitive Chain Abstraction. This figure focuses on the cycles containing *fr* edges. Acyclic graphs like *AbsCex* are abstract counterexamples: they may be concretized into real counterexamples or broken down into possible decompositions which are each checked in turn. Decompositions A and B are valid decompositions (subsets) of *AbsCex*, because they guarantee the edge from `p` to `q` in *AbsCex*. Decomposition C does not guarantee this edge, and is thus invalid and not considered further. Decomposition A strengthens the TC Abstraction enough to make the graph cyclic (and thus unobservable) as required. Decomposition B is valid but acyclic, so its abstraction needs to be refined further.

The TC Abstraction is the representation of an ISA-level chain of arbitrary length between two instructions i_1 and i_n as a single μhb edge between i_1 and i_n at the microarchitecture level. None of the intermediate instructions in the chain are explicitly modelled. *Abstract executions* are those executions where the TC Abstraction is used to represent an ISA-level chain. Meanwhile, *concrete executions* are those executions where all instructions and ISA-level edges are explicitly modelled; that is, where nothing is abstracted using the TC Abstraction. (Instructions in concrete executions are still symbolic.)

Figure 5.5 illustrates the TC Abstraction for ISA-level cycles containing the *fr* edge. The left of the figure represents all possible non-unary ISA-level cycles (i.e., the cycles containing more than one instruction[6]) that contain an *fr* edge. In these cycles, i_1 may be connected to i_n by a single ISA-level edge or by a chain of multiple ISA-level edges (the *transitive chain*). These cycles include the ISA-level cycle $po; rf; po; fr$ from Figure 5.1b (since it contains an *fr* edge), as well as an infinite number of other cycles containing *fr*. Using the TC Abstraction, any microarchitectural execution (as seen on the right) corresponding to such an ISA-level cycle represents the chain between i_1 and i_n by some μhb edge (the *transitive connection*) from a node of i_1 to a node of i_n. (The red μhb edge from i_n to i_1 is the mapping of the *fr* edge to the microarchitecture.) Thus, if the μhb graph on the right is verified to be unobservable for all possible transitive connections from i_1 to i_n, then the microarchitecture is guaranteed to be correct for all possible ISA-level cycles containing the *fr* edge. PipeProof automatically checks that the microarchitecture and ISA-level pattern support the TC Abstraction (Section 5.3.1) before using it to prove microarchitectural correctness.

The Transitive Chain Abstraction's capability to represent ISA-level chains of arbitrary length using a single transitive connection from the start to the end of the chain is the key insight underlying PipeProof's complete verification across all programs. This

[6]The number of unary cycles (those where an instruction is related to itself) is small, so PipeProof explicitly enumerates and checks them separately.

representation allows PipeProof to conduct unbounded verification while only explicitly modelling as many instructions as needed to prove microarchitectural correctness. The transitive connection models the effects of intermediate instructions in the ISA-level chain without explicitly modelling the instructions themselves, allowing for efficient modelling and verification of all possible ISA-level cycles. The TC Abstraction is both weak enough to apply to a variety of microarchitectures and also strong enough (with appropriate refinements discussed below) to prove microarchitectural MCM correctness.

5.2.5 Abstract Counterexamples

The TC Abstraction guarantees the presence of some μhb edge between the start and end of an ISA-level chain, such as that between i_1 and i_n in Figure 5.5. To prove microarchitectural correctness using the TC Abstraction, PipeProof must show that for each possible transitive connection between i_1 and i_n, all possible microarchitectural executions corresponding to the ISA-level cycles being checked are unobservable.

Figure 5.6 shows PipeProof's procedure for proving Theorem 2 on `simpleSC`. The figure focuses on the verification of ISA-level cycles containing at least one fr edge; the process is repeated for other types of cycles in the pattern (for SC, this equates to cycles containing po, co, or rf edges). For some transitive connections, like the one in the $NoDecomp$ case, the initial μhb graph of the abstract execution is cyclic. This proves the unobservability of all concrete executions represented by $NoDecomp$, as required for microarchitectural correctness.

In most cases, however, the initial abstract execution graphs will be acyclic, as is the case for $AbsCex$ in Figure 5.6. This is because the TC Abstraction's guarantee of a single μhb edge (the transitive connection) between the start and end of an ISA-level chain is necessarily rather weak in order to be general across *all* possible ISA-level chains that match a forbidden ISA-level pattern. The weakness of the guarantee is also necessary in order for the TC Abstraction to be general enough to support a variety of microarchitectures.

In abstraction refinement [CGJ+00], cases such as *AbsCex* that appear to violate the property being checked but contain an abstraction are called *abstract counterexamples*. They may correspond to concrete (real) counterexamples. They may also be *spurious*. When spurious, all concrete cases represented by the abstract counterexample are in fact correct.

In PipeProof, an abstract counterexample such as *AbsCex* represents two types of concrete executions. First, i_1 and i_n may be connected by a single ISA-level edge. On the other hand, i_1 and i_n may be connected by a chain of multiple ISA-level edges. To check whether an abstract counterexample is spurious or not, PipeProof conducts *concretization* and *decomposition* (the *refinement loop*) to handle the aforementioned two cases.

5.2.6 Concretization and Decomposition: The Refinement Loop

In the concretization step, PipeProof checks the case where i_1 is connected to i_n by a single ISA-level edge. PipeProof does so by replacing the transitive connection between i_1 and i_n with a single ISA-level edge that causes the resultant ISA-level cycle to match the forbidden ISA-level pattern. This concrete execution must be microarchitecturally unobservable. For example, when trying to concretize *AbsCex* in Figure 5.6, PipeProof checks the ISA-level cycle $po; fr$, then $co; fr$, then $rf; fr$, and finally $fr; fr$, since each of these are ISA-level cycles forbidden by SC that arise from replacing the transitive connection of *AbsCex* with a single ISA-level edge. If any of these concrete executions is found to be observable, then the microarchitecture is buggy and PipeProof returns the observable ISA-level cycle as a counterexample.

If executions where the transitive connection is replaced by a single ISA-level edge are found to be unobservable, PipeProof then inductively checks the case where i_1 is connected to i_n by a chain of multiple ISA-level edges through decomposition. PipeProof decomposes the transitive chain of $n-1$ ISA-level edges[7] into a transitive chain of $n-2$ ISA-level edges represented by a transitive connection, and a single concrete ISA-level edge. This also results in the explicit modelling of an additional instruction. The concrete ISA-level edge and instruction added by the decomposition are "peeled off" the transitive chain. The key idea behind decomposition of the transitive chain is that the explicit modelling of an additional instruction and ISA-level edge enforces additional microarchitectural constraints that may be enough to create a cycle in the graph. If a cycle is created, the decomposition is unobservable, completing the correctness proof for that case. If all possible decompositions of an abstract counterexample are found to be cyclic (unobservable), then the abstract counterexample is spurious and can be ignored. If any decomposition is found to be acyclic, then that decomposition constitutes an abstract counterexample itself, and concretization and decomposition are repeated for it. Decomposing the chain one instruction at a time improves efficiency by ensuring that PipeProof does not explicitly model more instructions than needed to prove microarchitectural correctness.

Figure 5.6 shows three (of many) possible decompositions of $AbsCex$. In Decomposition A, an rf edge has been peeled off the right end of the transitive chain. Peeling off the rf edge *strengthens* the abstraction by connecting node p to node r (an edge not present in $AbsCex$). This creates a cycle in the μhb graph, rendering the execution unobservable. This completes the correctness proof for this decomposition.

Decomposition B in Figure 5.6 shows a case where a *co* edge (rather than an rf edge) is peeled off the transitive chain. Furthermore, the *co* edge is peeled off the left end of the transitive chain rather than the right. Peeling off the *co* edge refines

[7]Here, n is an abstract parameter used for induction. It has no concrete value.

the abstraction, but this is still not enough to create a cycle in the μhb graph. Thus, Decomposition B is itself an abstract counterexample, and the process of concretization and decomposition will be repeated for it.

To ensure completeness of verification when decomposing transitive chains, PipeProof enumerates all possibilities for the concrete ISA-level edge that could be peeled off (using the procedure from Section 5.3.5) and for the transitive connection representing the remaining chain of length $n - 2$. Verifying a single decomposition is equivalent to verifying a subset of the executions of its parent abstract counterexample. As such, any valid decomposition must guarantee the transitive connections of its parent abstract counterexamples. Decompositions that violate this requirement do not represent executions that are modelled by their parent abstract counterexamples, and hence they are discarded.

For example, Decomposition C in Figure 5.6 is an invalid decomposition of $AbsCex$ because it does not guarantee the transitive connection of its parent $AbsCex$ (a μhb edge between nodes p and q) as Decompositions A and B do. PipeProof filters out any such decompositions that do not guarantee the transitive connections of their parent abstract counterexamples; it does not consider them further.

PipeProof alternates between peeling from the left and peeling from the right when inductively decomposing transitive chains. For example, Decomposition B was created by peeling from the left of $AbsCex$, so when concretization and decomposition is rerun for Decomposition B, the next ISA-level edge will be peeled from the right. PipeProof alternates in this manner because creating a cycle in the graph through decomposition often requires connecting more nodes to either side of the transitive connection.

5.2.7 Termination of the PipeProof Algorithm

In many cases, repeatedly peeling off instructions from the transitive chain strengthens the TC Abstraction enough to prove microarchitectural correctness. For the remaining

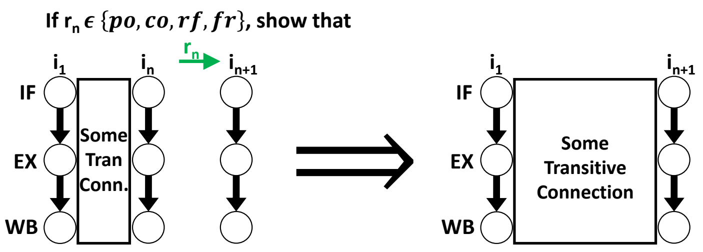

If r_n ∈ {po, co, rf, fr}, show that
r_n
i_1
i_n
i_{n+1}
IF
EX
WB
Some Tran Conn.
Some Transitive Connection

Figure 5.7: Graphical depiction of the inductive case of the TC Abstraction support proof for `simpleSC`. Extending a transitive chain by an additional instruction and ISA-level edge r_n should extend the transitive connection to the new instruction as well.

cases, PipeProof may be able to utilise user-provided *chain invariants* (Section 5.3.2) to abstractly represent infinite repeated peelings of a specific pattern of ISA-level edges and ensure termination of the refinement loop. For the SC and TSO case studies detailed in Section 5.5, peeling off a maximum of 9 instructions from the transitive chain was sufficient (along with chain invariants) to prove correctness of the microarchitectures.

PipeProof is not guaranteed to terminate in the general case, since it attempts to prove correctness across an infinite state-space (all possible programs). Prior work on enabling CEGAR for infinite-state systems also developed an algorithm that was not guaranteed to terminate [CFH$^+$03]. Thus, the absence of a termination guarantee in PipeProof is not unusual.

5.3 Supporting Proofs and Techniques

PipeProof's Microarchitectural Correctness proof relies on a number of supporting proofs and modelling techniques in order to prove correctness. This section explains these proofs and techniques, beginning with the TC Abstraction support proof (Section 5.3.1). This proof ensures that a microarchitecture and ISA-level pattern support the TC Abstraction, enabling it to be used in the Microarchitectural Correctness proof. Meanwhile, chain invariants (Section 5.3.2) are often required to ensure the termina-

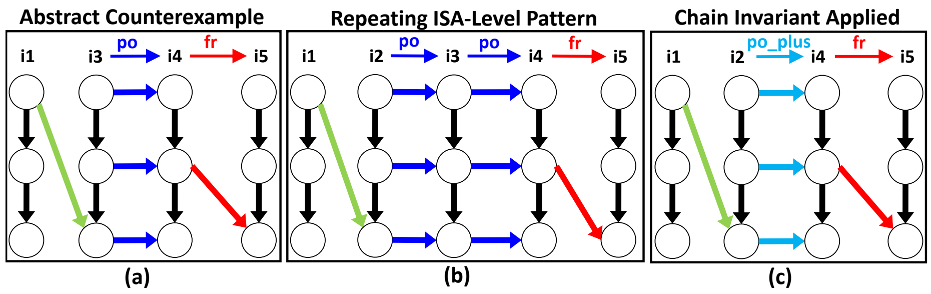


Figure 5.8: Peeling off edges from abstract counterexamples like (a) may cause repetitions of the same pattern, like *po* in (b). Naively continuing to peel off repeated edges in this manner may prevent the refinement loop from terminating. *Chain invariants* efficiently represent an arbitrary number of repetitions of such ISA-level patterns, as shown by *po_plus* in (c), allowing PipeProof's refinement loop to terminate.

```
Axiom "Invariant_poplus":
forall microop "i", forall microop "j",
HasDependency po_plus i j => AddEdge((i,Fetch),(j,Fetch),"") /\ SameCore i j.
```

Figure 5.9: Chain invariant for repeated *po* edges (i.e., *po_plus*) on the `simpleSC` microarchitecture.

tion of PipeProof's abstraction refinement loop. Theory Lemmas (Section 5.3.3) are required to constrain PipeProof's symbolic analysis to realisable microarchitectural executions. PipeProof must also use an over-approximation of microarchitectural constraints (Section 5.3.4) in order to guarantee soundness. Finally, Section 5.3.5 describes how PipeProof inductively generates ISA-level edges matching a pattern when decomposing transitive chains.

5.3.1 Ensuring Microarchitectural TC Abstraction Support

As discussed in Section 5.2.4, PipeProof uses the Transitive Chain (TC) Abstraction to represent the infinite set of ISA-level cycles that match a pattern like $cyclic(po \cup co \cup rf \cup fr)$. The TC Abstraction enables PipeProof to abstract away most of the instructions and ISA-level relations in these ISA-level cycles and represent them with a single μhb edge (the transitive connection) at the microarchitecture level.

The TC Abstraction is key to PipeProof's complete verification. However, to use the TC Abstraction in its Microarchitectural Correctness proof (Section 5.2), PipeProof must first ensure that the microarchitecture and ISA-level pattern being verified support the TC Abstraction. If the TC Abstraction cannot be proven to hold for a given microarchitecture, then PipeProof cannot prove the microarchitecture correct. PipeProof's verification is sound; it will never falsely claim that the TC Abstraction holds without proving it.

The theorem for microarchitectural TC Abstraction support is provided below, following the definition of an ISA-level subchain:

Definition 4. An ISA-level chain $r'_1; r'_2; r'_3 ...; r'_k$ is a *subchain* of an ISA-level cycle or chain $r_1; r_2; r_3 ...; r_n$ if $k < n$ and $r_i = r'_i$ for $i = 1$ to k. In other words, the subchain is the first k edges of the chain. For example, in Figure 5.1b, $po; rf; po$ is a subchain of the cycle $po; rf; po; fr$.

Theorem 3. If instructions i_A and i_B are connected by a transitive chain (i.e., a subchain of a forbidden ISA-level cycle), then there exists at least one μhb edge (the transitive connection) from a node of i_A to a node of i_B in all microarchitectural executions in which that subchain is present.

PipeProof tries to automatically prove Theorem 3 inductively for each microarchitecture and ISA-level pattern for which the TC Abstraction is used.

Base Case: In the base case, we need to show that any single ISA-level edge *isaEdge* that could be the start of the ISA-level chain to be abstracted guarantees a μhb edge between its source and destination instructions $i1$ and $i2$. For example, for `simpleSC`, PipeProof checks whether a po, co, rf, or fr edge between instructions $i1$ and $i2$ guarantees a μhb edge between them. As Figure 5.3 shows, the microarchitectural mappings of these ISA-level edges do indeed guarantee edges from $i1$ to $i2$ for `simpleSC`, so the base case passes.

Inductive Case: Figure 5.7 illustrates the inductive case for `simpleSC`. Given an ISA-level transitive chain between i_1 and i_n that implies a μhb transitive connection from i_1 to i_n, the inductive case must show that extending the transitive chain with an additional instruction i_{n+1} and ISA-level edge r_n matching the forbidden pattern extends the transitive connection. In other words, the inductive case must show that a μhb edge from some node of i_1 to some node of i_{n+1} exists.

If the combination of a transitive connection from i_1 to i_n and the microarchitectural mapping of r_n is not enough to guarantee a transitive connection from i_1 to i_{n+1}, this constitutes an abstract counterexample to Theorem 3. PipeProof then attempts to concretize and decompose the transitive chain between i_1 and i_n (as explained in Section 5.2.6) to discover whether the abstract counterexample is spurious or whether a concrete ISA-level chain violating Theorem 3 exists. ISA-level chains that fail Theorem 3 are henceforth referred to as *failing fragments*.

As in the Microarchitectural Correctness proof (Section 5.2), chain invariants (Section 5.3.2) are used to abstractly represent cases where an infinite number of edges could be peeled off without terminating. The abstraction refinement through decomposition continues until either the abstraction is strong enough to guarantee a transitive connection between i_1 and i_{n+1} in all cases (thus proving Theorem 3), or a failing fragment is found and returned to the user as failing the proof of Theorem 3.

Strength of Theorem 3: Theorem 3 is *stronger* than what the Microarchitectural Correctness proof (Section 5.2) needs. Theorem 3 requires the transitive chain to guarantee a transitive connection both when the transitive chain is part of a forbidden ISA-level cycle in the overall execution (as the Microarchitectural Correctness proof requires) *and* when it is not part of such an ISA-level cycle (as seen in Figure 5.7). This enables Theorem 3 to be proven by induction. As Figure 5.7 shows, the inductive case consists of adding an extra instruction and ISA-level edge to the case guaranteed by the induction hypothesis, resulting in a proof by induction.

On the other hand, proving the existence of a transitive connection only in the presence of a forbidden ISA-level cycle is not as straightforward. In the inductive case of such a proof, the induction hypothesis would guarantee a transitive connection for a chain between instructions i_1 and i_n only if the chain is part of a forbidden ISA-level cycle, similar to Figure 5.5. Extending this chain of length $n-1$ to a chain of length n (as required for an inductive proof) involves *removal* of one of the "loopback" edges connecting i_n to i_1 (fr in Figure 5.5). This is because a loopback edge connecting i_n to i_1 may not exist in arbitrary forbidden ISA-level cycles containing the extended transitive chain. If a loopback edge is removed, the induction hypothesis no longer guarantees a transitive connection between i_1 and i_n, and the proof cannot build on the guarantees for the chain of length $n-1$. In a nutshell, the induction hypothesis for such a proof is quite weak, so PipeProof cannot currently prove the necessary property, and attempts to prove the stronger Theorem 3 instead. This also means that some correct microarchitectures that do not satisfy Theorem 3 cannot be proven correct by PipeProof at present.

As a result of Theorem 3 being stronger than required, if a failing fragment is found, the microarchitecture may or may not be buggy. If the microarchitecture is buggy, PipeProof can generate an ISA-level cycle that exhibits the bug as a counterexample through its Cyclic Counterexample Generation procedure. This procedure checks all possible forbidden ISA-level cycles of length 1, then length 2, and so on for microarchitectural observability. At each iteration, if any of the cycles are microarchitecturally observable, the observable cycle is returned to the user as a counterexample. Otherwise, the procedure increases the size of the examined cycles by 1 and repeats the process.

5.3.2 The Need for Chain Invariants and their Proofs

When decomposing TC Abstractions instruction-by-instruction as outlined in Section 5.2.6, it is possible to peel off concrete ISA-level edges that match a repeating pattern, but for the abstraction to *never* be strong enough to prove the required property (Theorem 2 or 3). For example, Figure 5.8a shows an abstract counterexample to Theorem 3 on `simpleSC` where there is no μhb connection between instructions $i1$ and $i5$. When decomposing this abstract counterexample, it is possible to peel off a *po* edge from the transitive chain, as shown in Figure 5.8b, and still have no μhb connection between $i1$ and $i5$. In fact, one can continue peeling off *po* edges in this manner *ad infinitum*, while never being able to guarantee a μhb edge between $i1$ and $i5$. Such a case will result in the refinement loop of the Microarchitectural Correctness proof or TC Abstraction support proof being unable to terminate.

For refinement loops to terminate in such cases, PipeProof needs a way to efficiently represent such repeating patterns. To do so, PipeProof utilises user-provided[8] *chain invariants*. These chain invariants are additional μspec axioms which specify microarchitectural guarantees about repeated ISA-level edge patterns. Users can examine PipeProof's status updates to detect when the peeling off of repeated edge patterns is preventing termination of a refinement loop. This is a sign that the user needs to provide PipeProof with a chain invariant for the repeated edge pattern in question.

Figure 5.9 shows an example chain invariant for `simpleSC` that abstracts a chain of successive *po* edges as a single *po_plus*[9] edge. This invariant states that if two instructions i and j are connected by a chain of *po* edges of arbitrary length, then at the μhb level, i and j are guaranteed to be on the same core and to have a edge between their `Fetch` stages (which in turn implies edges between their `Execute` and `Writeback` stages due to the in-order `simpleSC` pipeline). An ISA-level chain of successive *po* edges of arbitrary length on `simpleSC` can then be abstractly represented by a single *po_plus* edge (and the guarantees of its invariant), as seen in Figure 5.8c.

[8]Future work could also use known invariant generation techniques to automatically discover invariants for a given concrete repeating ISA-level pattern. The search space of possible chain invariants is relatively small, and can be reduced further by restricting the search to specific invariant templates.

[9]The *plus* in *po_plus* is from Kleene plus (+).

PipeProof automatically searches for concrete ISA-level patterns that can be abstracted by user-provided invariants in each iteration of the refinement loop. The search for patterns matching available invariants is conducted edge by edge, similar to regex matching. Supported invariant patterns are repeated single edges (e.g., *po*) or repeated chains (e.g., $po;rf$). If PipeProof finds a concrete ISA-level pattern matching an invariant, it replaces the pattern with its invariant version. On subsequent decompositions, PipeProof's ISA Edge Generation procedure (Section 5.3.5) restricts the ISA-level edges that can be peeled off to those that cannot be subsumed within an adjacent chain invariant. For example, any edge peeled off from the right of the transitive chain in Figure 5.8c cannot be a *po* edge, as any such *po* edge is already subsumed within the *po_plus* edge between $i2$ and $i4$. This prevents edges matching an invariant pattern from being peeled off a transitive chain endlessly, allowing the refinement loop to terminate in such cases.

To help ensure verification soundness, PipeProof proves chain invariants inductively before using them in its proofs. If the proof of any chain invariant fails, PipeProof informs the user of the failure and does not proceed further. As an example of a chain invariant proof, consider Figure 5.9's invariant. PipeProof first checks the base case—whether a single *po* edge between two instructions i and j guarantees that they will be on the same core and have an edge between their `Fetch` stages. The *po* edge mapping and theory lemmas (Section 5.3.3) guarantee this. For the inductive case, PipeProof assumes that i and j are connected by a chain of a single *po* edge followed by a *po_plus* edge (i.e., a *po*-chain of arbitrary length), and that the invariant holds for the *po_plus* portion of the chain. It then checks if i and j are on the same core and have an edge between their `Fetch` stages. This property is guaranteed by the *po* edge mapping, theory lemmas, and the invariant from the induction hypothesis, completing the proof.

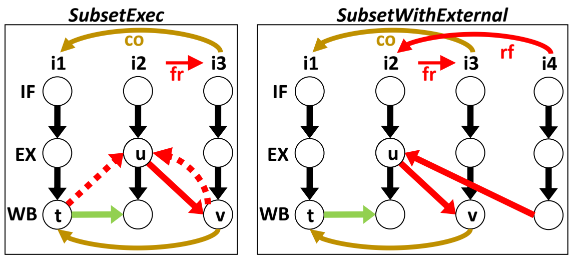


Figure 5.10: The load $i2$ in *SubsetExec* cannot read its value from the explicitly modelled stores $i1$ or $i3$ without adding one of the dotted edges and making the graph cyclic. This appears to make the execution unobservable. However, as shown in *SubsetWithExternal*, another instruction $i4$ outside the ISA-level cycle can source $i2$ while keeping the graph acyclic, making *SubsetWithExternal* an abstract counter-example. PipeProof must over-approximate microarchitectural constraints to account for instructions like $i4$ that are not explicitly modelled.

5.3.3 Theory Lemmas

The symbolic analysis conducted by PipeProof can allow inconsistent assignments to μspec predicates that are incompatible with *any* microarchitectural execution. For example, in any execution containing three instructions i, j, and k, if i and j have the same data (`SameData i j` is true) and j and k have the same data (`SameData j k` is true), then logically i and k must have the same data (`SameData i k` must be true). In other words, the `SameData` predicate is transitive. However, naive symbolic analysis will *not* require `SameData i k` to be true in such a case. Thus, to enforce such constraints, PipeProof provides a set of *Theory Lemmas*[10] for μspec predicates that is included in every call to PipeProof's solver. These constraints enforce universal rules (like the transitivity of `SameData`) that must be respected by *every* microarchitectural execution.

5.3.4 Over-Approximating to Ensure an Adequate Model

PipeProof verifies executions of an arbitary number of instructions while only modelling a small subset of those instructions and their constraints on the execution. Some of the instructions in an ISA-level cycle may be abstractly represented using the TC

[10]These lemmas are very similar to the lemmas produced by a theory solver in an SMT setup.

Abstraction or chain invariants, while other instructions in the execution that are not part of the ISA-level cycle are also not explicitly modelled. For its verification to be sound, PipeProof must ensure that the subset of an execution's constraints that it models is *adequate*: the subset must be an over-approximation of the constraints on the entire execution. In other words, it should never be the case that an execution is deemed to be unobservable when modelling only the subset of its constraints, but the execution is in fact observable.

For example, consider the abstract execution *SubsetExec* on `simpleSC` in Figure 5.10, where an ISA-level cycle is abstractly represented with the help of the TC Abstraction. Consider also the constraint (henceforth called `LoadSource`) on every ISA-level execution that for every load l which does not read the initial value of memory, there exists a store s for which $s \xrightarrow{rf} l$, corresponding to s being the store from which l reads its value. Instruction $i2$ in *SubsetExec* is a load (since it is the source of an *fr* edge), and so must satisfy the `LoadSource` constraint. If PipeProof attempted to satisfy `LoadSource` for $i2$ using only the explicitly modelled instructions, $i2$ could be sourced either from $i1$ (i.e., $i1 \xrightarrow{rf} i2$) or from $i3$ ($i3 \xrightarrow{rf} i2$). If sourcing from $i1$, the microarchitectural mapping of the rf edge adds a μhb edge from node t to node u, giving us a cycle in the μhb graph. Likewise, if $i3$ is used as the source, $i3 \xrightarrow{rf} i2$ maps to a μhb edge from node v to node u, once again creating a cycle in the graph. Thus, if the analysis only considered explicitly modelled instructions, it would deduce that all graphs for this case are cyclic (i.e., unobservable), and that this case need not be concretized and decomposed.

However, this reasoning would be incorrect. For instance, it is perfectly valid for an execution containing the ISA-level cycle from *SubsetExec* to have an additional instruction $i4$ that is not part of the ISA-level cycle but sources the value of $i2$ (i.e., $i4 \xrightarrow{rf} i2$). Figure 5.10 depicts this variant as *SubsetWithExternal*, which satisfies `LoadSource` while maintaining an acyclic graph. This indicates (correctly) that the

ISA-level cycle from *SubsetExec* is actually an abstract counterexample and must be concretized and decomposed.

To avoid unsoundly flagging executions such as *SubsetExec* as unobservable, PipeProof conservatively over-approximates by replacing every `exists` clause in the μspec with a Boolean `true`. This suffices to guarantee an adequate model, since `exists` clauses are the only μspec clauses whose evaluation can change from `false` to `true` when an additional instruction is explicitly modelled. In the case of *SubsetExec*, the over-approximation results in `LoadSource` always evaluating to true, ensuring that *SubsetExec* is treated as an abstract counterexample as required for soundness.

This over-approximation forces PipeProof to work only with a subset of the overall true microarchitectural ordering constraints; these may or may not be sufficient to prove the design correct. There exist microarchitectures for which this subset is not sufficient, and PipeProof currently cannot prove the correctness of those designs. However, the over-approximation of microarchitectural constraints is sufficient to prove the correctness of the designs in this chapter.

5.3.5 Inductive ISA Edge Generation

There are often an infinite number of ISA-level executions that can match a forbidden ISA-level pattern. Thus, PipeProof must reason about these ISA-level executions inductively to make verification feasible. PipeProof's refinement loop inductively models additional instructions through concretization and decomposition (Section 5.2.6 and Figure 5.6). As such, PipeProof must also inductively generate the possible ISA-level relations that could connect these modelled instructions such that the overall execution matches the forbidden ISA-level pattern being checked.

Given an ISA-level pattern *pat*, PipeProof's ISA Edge Generation procedure returns all possible choices $(edge, remain)$, where *edge* is a possible initial or final edge of *pat*, and *remain* is the part of *pat* that did not match *edge*. If peeling from the left of a transitive chain, the procedure returns cases where *edge* is an initial edge. If peeling from the right, the procedure returns cases where *edge* is a final edge.

For example, if decomposing a transitive chain representing the pattern $(po \cup co); rf; fr$, the ISA Edge Generation procedure would return $(po, (rf; fr))$ and $(co, (rf; fr))$ if peeling from the left, so either *po* or *co* could be peeled off. Likewise, if peeling from the right, the procedure would return $(fr, ((po \cup co); rf))$, so only *fr* could be peeled off.

5.4 PipeProof Optimizations

To improve PipeProof's verification performance and make its verification feasible for more designs, I implemented two optimizations: the *Covering Sets Optimization* and *Memoization*. This section explains these optimizations.

5.4.1 Covering Sets Optimization

The TC Abstraction guarantees at least one transitive connection between the start and end of an ISA-level chain that it represents. Thus, PipeProof needs to verify correctness for *each* possible transitive connection when using the TC Abstraction to represent an ISA-level chain. As seen in Figure 5.6, a new set of transitive connections comes into existence each time a transitive chain is decomposed. This can quickly lead to a large number of cases to consider. Even the `simpleSC` microarchitecture has 9 possibilities (3 nodes $*$ 3 nodes) for transitive connections between any two instructions. To mitigate this case explosion, I developed the *Covering Sets Optimization* to eliminate redundant transitive connections.

The key idea behind the Covering Sets Optimization is that if in a given scenario, a and b are possible transitive connections, and every μhb graph containing a also contains b, then it is sufficient to just check correctness when b is the transitive connection. In other words, *b covers a*. For example, *AbsCex* in Figure 5.6 has a transitive connection between nodes p and q. This transitive connection covers other possible transitive connections such as the one from p to r used in *NoDecomp*. This is because there is no possible μhb graph satisfying the microarchitectural axioms that contains an edge from p to r without also having an edge from p to q (by transitivity). Given a set of transitive connections *conns* for a given scenario along with all other scenario constraints, the Covering Sets Optimization eliminates transitive connections in *conns* that are covered by other transitive connections in the set. This optimization significantly improves PipeProof runtime (details in Section 5.5).

5.4.2 Eliminating Redundant Work Using Memoization

Figure 5.6 shows PipeProof's procedure for proving that `simpleSC` is a correct implementation of SC. PipeProof first checks that all ISA-level cycles containing *fr* are microarchitecturally unobservable, and then does the same for cycles containing rf, *po*, and *co*. However, there is notable overlap between these four cases. For example, the ISA-level cycle $po; rf; po; fr$ from Figure 5.1b contains *po*, rf, and *fr* edges. A naive PipeProof implementation would verify this cycle (directly or indirectly through the TC Abstraction) at least 3 times: once as a cycle containing *po*, once as a cycle containing rf, and once as a cycle containing *fr*. The redundant second and third checks of the cycle can be eliminated.

PipeProof filters out cases that have already been verified by restricting the edges that can be peeled off during decomposition. For example, if all ISA-level cycles containing *fr* have been verified for `simpleSC`, then when checking all ISA-level cycles containing *po*, *fr* edges should be excluded from the choices of edges to peel off. This is because peeling off an *fr* edge would turn the ISA-level cycle being considered into a cycle containing *fr* (which has already been verified).

Stated formally, given an ISA-level MCM property $acyclic(r_1 \cup r_2 \cup ... \cup r_n)$, if all ISA-level cycles containing r_i have been verified $\forall i < j$, then the only choices for edges to peel off when verifying cycles containing r_j should be $\{r_j, r_{j+1}, ..., r_n\}$. This optimization enables my TSO case study (Section 5.5) to be verified in under an hour.

5.5 Methodology, Results, and Discussion

PipeProof is written in Gallina, the functional programming language of Coq [Coq04]. PipeProof reuses PipeCheck's μspec parsing and axiom simplification [LSMB16], and extends PipeCheck's solver to be able to model and verify executions of symbolic instructions. PipeProof's Gallina code is extracted to OCaml using Coq's built-in extraction functionality. This OCaml code is then compiled into a standalone PipeProof binary that can be run by a user.

I ran PipeProof on two microarchitectures. The `simpleSC` microarchitecture has a 3-stage in-order pipeline and Store→Load ordering enforced. The `simpleTSO` microarchitecture is `simpleSC` with Store→Load ordering relaxed for different addresses. I verified `simpleSC` against the SC ISA-level MCM, while `simpleTSO` was verified against the TSO ISA-level MCM. The overall specification of TSO consists of two properties: $acyclic(po\_loc \cup co \cup rf \cup fr)$ and $acyclic(ppo \cup co \cup rfe \cup fr \cup fence)$ [AMT14]. The *po_loc* relation models same-address program order, while *ppo* (preserved program order) relates instructions in program order except for Store→Load pairs (which can be reordered under TSO). The *rfe* (reads-from external) edge represents when a store sources a load on another ("external") core, and *fence* relates instructions separated by a fence in program order. In the case of TSO, fences modelled by the $fence$ relation enforce ordering between stores before the fence in program order and loads after the fence in program order.

Component	`simpleSC`	`simpleSC` (w/ Covering Sets)	`simpleSC` (w/ Covering Sets + Memoization)	`simpleTSO` (w/ Covering Sets)	`simpleTSO` (w/ Covering Sets + Memoization)
Chain Invariant Proofs	0.008 sec	0.01 sec	0.008 sec	0.5 sec	0.5 sec
TC Abstraction Support Proofs	2.8 sec	0.9 sec	0.9 sec	71.1 sec	67.3 sec
Microarch. Correctness Proofs	223.1 sec	35.5 sec	18.2 sec	19813.8 sec	2379.5 sec
Total Time	**225.9 sec**	**36.4 sec**	**19.1 sec**	**19885.4 sec**	**2449.7 sec**

Table 5.1: PipeProof runtimes for `simpleSC` and `simpleTSO` with and without Covering Sets and Memoization.

Experiments were run on an Ubuntu 16.04 machine with an Intel Core i7-4870HQ processor. Table 5.1 breaks down PipeProof runtimes for five cases. For `simpleSC`, the evaluated configurations are: (i) using vanilla PipeProof algorithms, (ii) with the Covering Sets Optimization (Section 5.4.1), and (iii) with Covering Sets and Memoization (Section 5.4.2). For `simpleTSO`, the evaluated configurations are: (iv) using Covering Sets, and (v) with Covering Sets and Memoization. (`simpleTSO` was infeasible without the Covering Sets Optimization.) PipeProof proves the correctness of `simpleSC` in under four minutes using vanilla PipeProof algorithms. The Covering Sets Optimization brings runtime down to under a minute, and Memoization reduces runtime further to under 20 seconds. Meanwhile, proving that `simpleTSO` correctly implements TSO takes just over five and a half hours with the Covering Sets Optimization. With the addition of Memoization, `simpleTSO` is verified in under 41 minutes.

While runtimes under an hour are quite acceptable, the verification of `simpleTSO` takes more time than the verification of `simpleSC` because TSO's additional relations increase the number of possibilities for ISA-level edges that can be peeled off a transitive chain. This has a multiplicative effect on the number of cases that need to be verified; each peeled-off instruction may require verification across many transitive connections, each of which may require further instructions to be peeled off. Nevertheless, with the help of its optimizations, PipeProof's verification of `simpleTSO` in under an hour shows that complete automated MCM verification of microarchitectures can indeed be tractable.

With regard to chain invariants, verifying `simpleSC` required one invariant (*po_plus*) to be provided to model repeated *po* edges. Meanwhile, verifying `simpleTSO` required five invariants, for repetitions of *ppo*, *fence*, *po_loc*, *ppo* followed by *fence*, and *fence* followed by *ppo*.

PipeProof's detection of microarchitectural bugs was quite fast. As an example, I introduced a flaw into `simpleSC` relaxing Store→Load ordering. PipeProof produced a counterexample to that flaw in under a second (both with and without the Covering Sets optimization). Similarly, if Store→Load ordering for the same address was relaxed on `simpleTSO`, the bug was detected in under 2 seconds from the beginning of the check of the relevant ISA-level pattern.

To scale verification performance to more complicated microarchitectures, the implementation of PipeProof's algorithm (Section 5.2) can be parallelised. The only dependency in the algorithm is that a given abstract execution (such as $AbsCex$ from Figure 5.6) must be checked before any concretizations or decompositions of it are checked. Apart from this dependency, each abstract or concrete execution can be checked independently of the others, making the algorithm *highly* parallelizable and well-suited to the use of multicore machines and clusters to improve performance. Adding RealityCheck's support for modularity to PipeProof would also improve its scalability.

5.6 Related Work

Mador-Haim et al. [MHAM11] established a bound on litmus test completeness when comparing certain consistency *models*, but it is still unknown how to detect whether a test suite is complete with respect to a given parallel system *implementation*. As such, there is no way to tell whether passing a suite of litmus tests means that a design is correct for all programs. PipeProof sidesteps this problem by conducting complete verification of all possible programs on a microarchitecture without using litmus tests at all.

Chatterjee et al. [CSG02] verify operational models of processor implementations against operationally specified ISA-level MCMs. Their approach has two main steps.

The first step creates an abstract model of the microarchitectural implementation by abstracting away the external memory hierarchy, and verifies it by checking a refinement relation between the two models using model checking. The second step verifies this abstract model against an ISA-level MCM specification using theorem-proving. They only verify small instances (restricted to two processors, addresses, and values), while PipeProof's verification is *complete* across different core counts, addresses, and values. They also target verification of operational models, while PipeProof targets axiomatic models. Finally, they handle visibility order specifications, whereas PipeProof uses more general MCM specifications such as the acyclicity of certain ISA-level edge patterns.

The only complete proofs of microarchitectural MCM correctness that have been conducted in prior work are those of the Kami project [VCAD15, $\text{CVS}^{+}17$]. However, Kami utilises the Coq interactive theorem prover [Coq04] for its proofs, which requires designers to know proof techniques and requires manual effort. This is not amenable for many computer architects. In contrast, PipeProof *automatically* proves microarchitectural MCM correctness when provided with an ISA-level MCM specification, μspec axioms, mappings, and invariants. My development of PipeProof thus opens up how and by whom such techniques can be used.

Prior work on automated refinement checking (e.g. Burch and Dill [BD94], Chatterjee et al. [CSG02]) has used operational models. In contrast, PipeProof uses axiomatic models. To my knowledge, PipeProof is the first ever automated refinement checking methodology and tool for pairs of axiomatic models of executions. As such, PipeProof furthers the state of the art in refinement checking in addition to its contributions to MCM verification.

5.7 Chapter Summary

The MCM verification of a hardware design must be *complete* (i.e., must verify across all possible programs) in order to guarantee the correct execution of parallel programs on that hardware. However, prior microarchitectural MCM verification approaches either required manual proofs or only conducted bounded or litmus-test-based verification.

In response, this chapter presents PipeProof, the first methodology and tool for automated all-program verification of axiomatic microarchitectural ordering specifications with respect to axiomatic ISA-level MCM specifications. PipeProof can either automatically prove a specified microarchitecture correct with respect to its ISA-level MCM or it can inform the user that that the microarchitecture could not be verified, often providing a counterexample to illustrate the relevant bug in the microarchitecture. Furthermore, to my knowledge, PipeProof is the first ever automated refinement checking methodology and tool for pairs of axiomatic models of executions, so it makes advances in refinement checking as well.

PipeProof's novel Transitive Chain Abstraction allows it to inductively model and verify all microarchitectural executions of all possible programs. This enables efficient yet complete microarchitectural MCM verification; PipeProof is able to prove the correctness of microarchitectures implementing SC and TSO in under an hour. With PipeProof, architects no longer have to restrict themselves to manual proofs of correctness of their designs in interactive proof assistants to achieve all-program MCM verification. Instead, they can use PipeProof to automatically prove microarchitectural MCM correctness for all possible programs, addresses, and values.

Chapter 6

Progressive Automated Formal Verification

And one man in his time plays many parts,
His acts being seven ages.

—William Shakespeare
As You Like It

Validation[1] of a computing system is essential to ensuring its correctness, and can form a substantial part of the time spent developing the system. Traditionally, validation is conducted after the implementation of a component or system has been completed, and is carried out using testing-based methods. However, conducting formal verification earlier in the development timeline can have important benefits, including the ability to detect bugs earlier. At the same time, early-stage verification is not sufficient on its own to ensure system correctness. Engineers may still make mistakes when implementing a verified early-stage design, and post-implementation verification is necessary to catch these bugs. Furthermore, linking early-stage verification methodologies to post-implementation verification efforts can decrease verification overhead, thus reducing overall verification time. Verification methodologies for inter-

[1]Recall that in this dissertation, I use the term *validation* to refer to techniques for checking or ensuring the correctness of a system, including both testing-based methods and formal verification. Meanwhile, I use the term *verification* to refer to verification using formal methods.

mediate points in the development timeline may be necessary to aid in this linkage, as early-stage and post-implementation verification methodologies are often very different and hard to connect to each other.

This chapter presents a verification flow called *Progressive Automated Formal Verification*[2] which advocates conducting verification at multiple points in the development timeline and linking the verification methods at each stage together. Progressive verification has multiple benefits, including earlier detection of bugs, reduced verification overhead, and reduced overall development time. The work in Chapters 3, 4, and 5 can be combined to form a Progressive Automated Formal Verification flow for MCM properties in parallel processors.

This chapter begins by covering verification in a traditional development flow (Section 6.1). It then covers verification approaches for early-stage designs (Section 6.2), post-implementation verification approaches (Section 6.3), and verification approaches for evolving and detailed designs (Section 6.4). This is followed by an explanation of the Progressive Automated Formal Verification flow (Section 6.5), which utilises all three of these verification approaches. Section 6.5's explanation uses the work in Chapters 3, 4, and 5 as an example progressive verification flow. Section 6.6 then summarises the chapter.

6.1 Testing and Verification in a Traditional Development Flow

Figure 6.1 shows a traditional development flow for a hardware or software system. The system is first designed, then implemented, and subsequently tested and/or formally verified. Testing is the dominant validation technique in industry today, although industry usage of formal methods is increasing [RCD+16, NRZ+15]. Formal

[2]sometimes shortened to "progressive verification" for brevity.

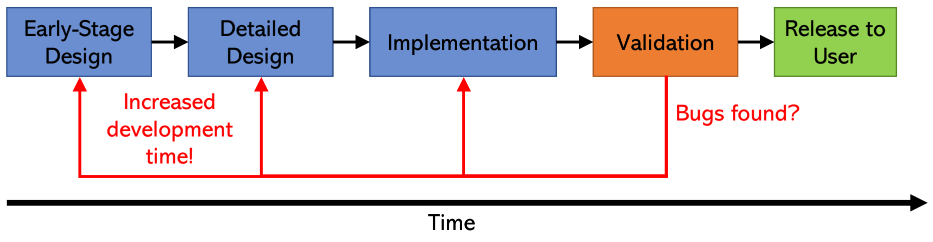


Figure 6.1: A traditional system development flow. Validation (testing and/or verification) begins after implementation commences, so design bugs (and sometimes even high-level specification bugs) are only found during post-implementation verification. The discovery of design bugs or high-level specification bugs may necessitate a redesign, resulting in the loss of development time spent creating an implementation of the buggy design.

methods are superior to testing-based methods as they are adept at checking all possibilities for a given scenario, e.g., PipeProof can prove microarchitectural MCM correctness across all executions of all programs. Bounded verification methods cannot guarantee correctness in all possible scenarios, but are complete for the scenarios that they do check. For example, PipeCheck can prove microarchitectural MCM correctness for a suite of litmus tests. Meanwhile, testing-based methods cannot guarantee verification across all programs, or even across all executions for a single program on nondeterministic systems like today's multiprocessors.

The design of a system may have multiple phases. For instance, design of a processor may begin with a high-level design that becomes more detailed as development continues. The initial high-level design might include data like the number of pipeline stages, whether they are in-order or out-of-order, and the number and size of caches. A subsequent detailed design would delve deeper into the inner workings of the various components, including the protocol for memory transactions between processors, caches, and main memory, as well as the input/output behaviour of queues and buffers. The detailed design would also include a fine-grained structure for each pipeline stage.

Once a design is fleshed out, the hardware or software system is implemented according to that design. This is followed by testing and/or verification of the implementation. Continuing with the processor example, a processor is usually first implemented in RTL like Verilog, which is where testing and/or verification typically begins. Once the processor's RTL has been written, it is tested and/or verified. Engineers then progress to the lower-level details of implementation, like how the transistors comprising the processor will be laid out on the chip.

The key takeaway here is that in traditional development flows like Figure 6.1, verification typically does not begin until implementation has been at least partially completed. However, beginning verification earlier in the development timeline can have several benefits, as discussed next.

6.2 The Benefits of Early-Stage Design-Time Verification

Bugs in systems can arise in their designs, their implementations, or even their high-level specifications. Design bugs are cases where a system's design is incorrect and does not correspond to the high-level specification's requirements. If a design bug exists, the system will not function correctly with respect to higher-level requirements even if the implementation faithfully implements the design. An example design bug would be the use of an out-of-order pipeline (without enforcing ordering through an alternative mechanism) in a processor design that has an MCM of SC. (The ISA-level MCM functions as the high-level specification in this example.) Meanwhile, implementation bugs occur when engineers unintentionally introduce flaws during system implementation, causing the implementation to not match the design. For instance, if the RTL of a cache only returns the lower 30 bits at a memory address in response to a processor's 32-bit load operation, this would constitute an implementation bug.

Bugs may also occur in high-level specifications themselves. For instance, designers may choose an ISA-level MCM for their processor that they believe allows the microarchitectural optimizations they wish to include. However, subsequent verification may discover that the MCM is too strong and forbids some of their optimizations. The MCM may also be too weak to interface with high-level languages.[3] If the ISA for the processor has not yet been released, the designers may decide to change its MCM to fix these issues, rather than modify their designs. Thus, a given issue may be classified as a design bug or a high-level specification bug depending on designer intent, provided that the high-level specification has not been set in stone.

While implementation bugs can only be caught during or after implementation, *design bugs and high-level specification bugs can be caught during the design phase itself.* Validation approaches based on dynamic testing require an implementation in order to validate a system, and so are not a good fit for design-time validation. However, formal verification can be conducted on a model of the design and so is well-suited for this purpose. For instance, PipeCheck [LPM14, LSMB16] (Section 2.4) and PipeProof (Chapter 5) are examples of early-stage verification tools for MCM properties in hardware designs. Likewise, CheckMate [TLM18a] is an example of an early-stage verification tool for hardware security properties. Early-stage verification has generally not been conducted for hardware designs in prior work, except for a few specific types of features like cache coherence protocols [PNAD95, ZLS10, ZBES14]. (The work of Vijayaraghavan et al. [VCAD15] is a notable exception to this rule, and is covered in Section 2.3.4.) Early-stage verification is more prominent for software systems, e.g., Memalloy [WBSC17] and distributed protocol verification [PLSS17].

In a traditional development flow like Figure 6.1, design bugs, implementation bugs, and sometimes even high-level specification bugs are discovered during post-implementation verification. Thus, if the design or high-level specification contained a bug, engineers would spend time and effort implementing the buggy design, only

[3]TriCheck [TML+17] enables users to verify (on a per-test basis) that their choice of ISA-level MCM is compatible with their choices of microarchitectural optimizations and the requirements of high-level language MCMs.

to have to re-implement the relevant parts of the system once a re-design fixed the design or high-level specification bug. The development time spent creating the implementation of the incorrect design (which may be a significant duration) is thus wasted. Alternatively, engineers may eschew a re-design and implement a heavy-handed fix that notably reduces the capabilities of the system. In either case, the outcome is undesirable.

Using approaches like PipeCheck, PipeProof, and CheckMate for early-stage verification helps catch design and high-level specification bugs right at design time, so they can be fixed before implementation commences. This eliminates the time that would otherwise be spent creating implementations of incorrect designs, thus reducing overall development time.

In addition, early-stage verification can reduce the overhead of post-implementation verification (Section 6.3) if the two approaches are linked to each other. For example, if a μspec model of a design is verified through PipeProof as correctly implementing its MCM, then the eventual RTL implementation of that design need only ensure that it satisfies the individual axioms of the μspec model to ensure MCM correctness. RTLCheck (Chapter 3) enables such post-implementation verification for litmus tests. In this scenario, the low-level RTL implementation no longer needs to be verified against the ISA-level MCM directly. Such verification would be harder than verifying against μspec axioms, as the ISA-level MCM is a high-level property which is quite far removed from the low-level details of an RTL implementation.

Despite its advantages, design-time verification is not a panacea. Even if a design has been formally verified, engineers may introduce implementation bugs as they implement and refine the design. Post-implementation verification (discussed next) is essential to ensure the system's correctness before its release to the end user.

6.3 The Need for Post-Implementation Verification

Post-implementation validation is the traditional method of ensuring hardware and software system correctness today. Such validation is critical to ensuring that the implementation released to end users is in fact correct. There has been a plethora of work on post-implementation formal verification. Some of this work uses automated approaches like model checking, e.g., CBMC [CKL04] for checking properties of C programs. Other approaches may include manually proving system correctness in a proof assistant like Coq and then extracting an implementation, e.g., Kami [CVS+17] for hardware verification (Section 2.3.4). Sections 2.3.2, 2.3.4, and 3.10 cover the prior work on post-implementation verification that is most relevant to this dissertation. RTLCheck (Chapter 3) is a methodology and tool for automated post-implementation MCM verification.

While industry usage of formal methods is increasing [RCD+16, NRZ+15], testing-based methods remain the primary method of post-implementation validation in industry today. These testing-based methods cannot ensure comprehensive verification coverage, and lead to bugs slipping through validation into the released product. Such bugs can have serious consequences, including Internet outages [Str19] and car crashes [Lee19]. Current industry practices have proven insufficient for catching bugs in recent years, ranging from security vulnerabilities like Meltdown [LSG+18] and Spectre [KHF+19] to network errors [BGMW17] to MCM issues [ARM11].

While software is relatively easy to patch when bugs are found, hardware implementations are difficult or impossible to modify after their release. As a result, post-release fixes for hardware bugs are often heavy-handed and can notably impact processor performance. One such bug was a TLB[4] issue (Erratum #298) in AMD

[4]Translation Lookaside Buffer, a cache for virtual to physical page mappings.

Phenom processors [AMD12] that could lead to data corruption and loss of coherence. AMD initially[5] had to fix this bug with a BIOS patch that prevented TLBs from looking in the cache for page table entries, which caused a slowdown of about 10% [Shi08]. Likewise, the transactional memory functionality in Intel Haswell processors was found to be buggy after their release [Hac14]. To work around the issue, Intel disabled the transactional memory functionality in Haswell processors, reducing their capabilities.

Post-implementation verification should use formal methods as much as possible in order to ensure better verification coverage, thus reducing or eliminating bugs in the released product. This is especially important for hardware due to its inability to be modified after release. In addition, the formal methods used should be automated approaches like model checking (Section 2.2.1). This will allow hardware and software engineers who do not have deep formal methods expertise to verify the systems by themselves, rather than having to rely on the relatively small number of formal methods experts. For these reasons, the Progressive Automated Formal Verification flow (Section 6.5) proposed by this dissertation recommends the use of automated formal verification.

6.4 Verification at Intermediate Points in the Development Timeline

Early-stage design-time verification and post-implementation verification are both key to ensuring a system's correctness. However, a system evolves significantly as it progresses from early-stage design through to a final detailed design. Design bugs may be introduced at any point in this evolution. It is important to detect these bugs as early as possible so as to minimise the time that engineers spend creating

[5]A subsequent stepping of the Phenom processor included a hardware fix for the issue [AMD12, Shi08].

implementations of incorrect designs. This necessitates the continued verification of designs as they evolve and become more detailed.

Early-stage verification tools may not be a good fit for verifying evolving designs, necessitating the creation of verification tools specifically for such evolving designs. For instance, as Section 4.2.2 covers, the monolithic MCM verification of early-stage verification tools like PipeCheck and PipeProof does not scale to detailed models of large designs. Furthermore, their monolithic specification format is a bad fit for the distributed nature of processor design. Monolithic specifications also make it hard to replace pieces of the design specification over time as the design evolves. As a result, these tools are insufficient for MCM verification of evolving and detailed designs. RealityCheck (Chapter 4) was created specifically to be capable of efficiently verifying designs as they evolve and become more detailed.

Verification tools for intermediate points in the development timeline can also facilitate the linkage of early-stage verification to post-implementation verification. As Section 6.2 covers, the linkage of early-stage verification to post-implementation verification can reduce the overhead of post-implementation verification. However, early-stage designs are quite far removed from the low-level realities of implementations, making them hard to link to each other. For instance, the early-stage MCM verification of a tool like PipeProof is based on omniscient μspec axioms (Section 4.2.2) that do not reflect the structural modularity of RTL implementations. As Section 4.7 covers, SVA assertions generated from such omniscient μspec axioms using a tool like RTLCheck (Chapter 3) are limited in their scalability because they must be evaluated over the entire RTL implementation. On the other hand, RealityCheck's μspec++ axioms take into account the design's structural modularity and are scoped to individual modules. As a result, SVA assertions generated from these μspec++ axioms will be scoped to individual modules as well. These generated assertions will thus only need

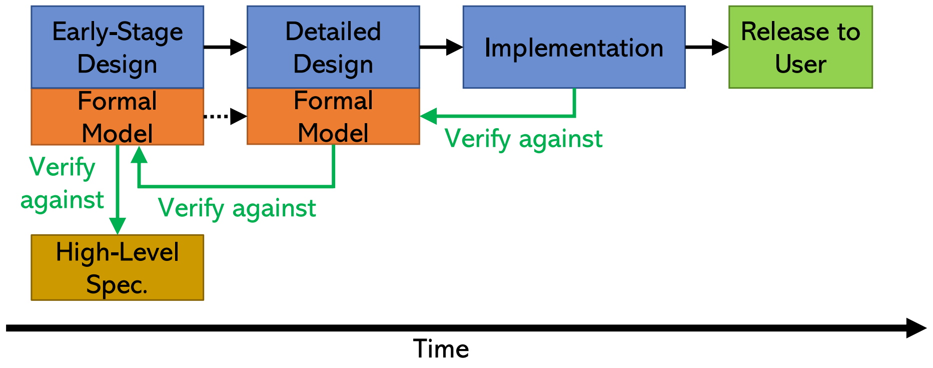


Figure 6.2: An idealised version of the Progressive Automated Formal Verification flow. Verification is conducted at multiple points in the development of the system (starting at early-stage design), and the verification methods at each stage are linked to each other. The flow can have more or fewer stages depending on the needs of the system and the choices of the engineers involved. Progressive verification provides earlier detection of bugs, reductions in development time and verification overhead, and the strong correctness guarantees of formal verification.

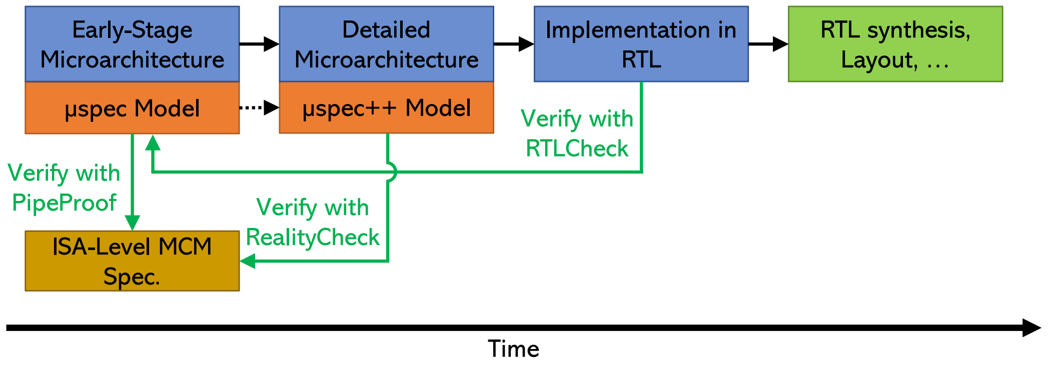


Figure 6.3: A Progressive Automated Formal Verification flow for MCM properties in parallel processors. PipeProof (Chapter 5) provides MCM verification of early-stage designs, RealityCheck (Chapter 4) provides MCM verification of detailed designs, and RTLCheck (Chapter 3) provides post-implementation MCM verification.

to be evaluated over part of the RTL implementation, significantly improving their scalability.

6.5 Progressive Automated Formal Verification

Early-stage, intermediate, and post-implementation verification all have their own advantages and disadvantages. If all of these verification methodologies could be used in concert with each other and linked together, it would make system verification more thorough and efficient. This is the key idea underlying *Progressive Automated Formal Verification*, a verification flow proposed by this dissertation. The rest of this section explains the details of Progressive Automated Formal Verification.

Figure 6.2 shows an idealised flow for Progressive Automated Formal Verification. Meanwhile, Figure 6.3 shows the specific instance of the progressive verification flow that applies to MCM properties in parallel processors. This instance of the flow utilises the work in Chapters 3, 4, and 5, i.e., the RTLCheck, RealityCheck, and PipeProof tools respectively.

As Figure 6.2 shows, progressive verification begins verification right at the point of early-stage design, in contrast to traditional verification (Figure 6.1). During early-stage design, formal models of the system are created and automatically verified against higher-level requirements. For example, if verifying hardware MCM properties, the verification task at this stage would be to create a μspec model of the processor and verify it against the ISA-level MCM using PipeProof (Chapter 5), as shown in Figure 6.3.

As the design evolves and becomes more detailed, progressive verification requires that the formal model evolve along with it to reflect such detail, as shown in Figure 6.2. The evolution of the design will likely also coincide with more people being involved with it, making the formal model something that a large number of individuals will be involved with editing. Verification should continue to be conducted as the design (and its model) evolve to ensure that bugs are not introduced while adding the additional detail. The detailed design can be verified either against the early-stage design or against the higher-level properties being verified, depending on the preference of the engineers and the relative difficulty of the two verification tasks. The modelling and verification frameworks used should facilitate the evolution of the design as it progresses.

In the case of MCM verification, the RealityCheck framework (Chapter 4) fulfills the requirements of modelling and verification for evolving and detailed designs, as seen in Figure 6.3. Initially, users can create a coarse-grained μspec++ model of the processor in RealityCheck and compare it (up to a bound) to the model verified using PipeProof. They may also verify it directly against ISA-level litmus tests, as Figure 6.3 shows (and as explained in Chapter 4). Then, as the design of various components is fleshed out, the specifications for those components can be replaced with their more detailed versions. Interface verification (Section 4.4) can be used to ensure that the detailed specification is correct with respect to the component's requirements. The modularity of RealityCheck also allows different teams to modify or replace the specification of their components (and verify them against their interfaces) independently of other teams. This ensures that the distributed nature of the hardware design process is not impeded by the use of RealityCheck.

In a progressive verification flow, early-stage and intermediate-stage verification reduce the verification load on post-implementation verification. Instead of having to verify the implementation directly against higher-level properties, engineers and developers only need to verify their implementations against the formal specifications whose correctness has been checked by verification at earlier stages. In the case of progressive MCM verification (Figure 6.3), RTLCheck provides post-implementation verification. Using RTLCheck, the processor's Verilog is formally verified against μspec axioms (on a per-test basis) whose composition would ideally have been previously proven correct using PipeProof. This is notably easier than trying to verify the low-level Verilog directly against the high-level ISA MCM specification, as they are quite far removed from each other.

Ideally, the Verilog would be checked against SVA properties translated from a RealityCheck μspec++ model, as the μspec++ would be more detailed and easier to relate to RTL. Furthermore, the modularity of μspec++ axioms would enable scalable per-component verification of the RTL. Section 4.7 provides further details. The linkage of RTLCheck directly to μspec axioms rather than to μspec++ axioms from a RealityCheck model is due to RealityCheck being developed chronologically after RTLCheck.

Progressive verification has a number of benefits, as listed below:

- **Earlier Detection of Bugs:** Progressive verification begins verification right at the point of early-stage design, and continues to verify the design as it evolves. Thus, design bugs and high-level specification bugs are caught under progressive verification before implementation begins, in contrast to traditional verification (Figure 6.1) where they are caught post-implementation.

- **Reduced Overall Development Time:** Progressive verification's early detection of design bugs and high-level specification bugs means that engineers do not spend time creating implementations of incorrect designs. This can reduce overall development time when compared to a traditional development flow.

- **Reduced Verification Overhead:** Many properties are easier to verify at a higher level of abstraction than at a lower level. For instance, RTLCheck runtimes (Section 3.9) far exceed those of PipeProof (Section 5.5) for similar designs. This is despite PipeProof verifying MCM properties for microarchitectural designs across all programs and RTLCheck only verifying μspec axioms for litmus tests. Thus, verifying properties at a higher level of abstraction when possible will reduce the overhead of verification. This is especially critical for hardware design, where verification costs now dominate total hardware design cost [Fos15]. Progressive verification begins verification at early-stage design, ensuring that as much verification as possible is conducted at high levels of abstraction and thus at lower overhead.

- **Strong Correctness Guarantees:** The formal methods utilised at each stage of progressive verification ensure thorough verification of a system's design and implementation for the properties being verified. The formal nature of the verification enables engineers to provide strong correctness guarantees about the systems they build, as opposed to the weaker guarantees of testing-based methods. Systems verified using a progressive flow will be released to users with minimal or no bugs.

The work in Chapters 3, 4, and 5 can be combined to create a concrete instance of the progressive verification flow for MCM properties in parallel processors, as Figure 6.3 shows. This dissertation thus serves as a reference point for the future development of progressive verification flows for other types of properties and systems, such as for hardware security verification or compiler verification (see Section 7.3.3).

The verification methodologies used in a progressive flow would ideally be complete, i.e., they would verify across all possible programs. In the absence of automated all-program verification approaches for a given point in the development timeline, bounded verification approaches can be used instead. In the case of progressive MCM verification (Figure 6.3), PipeProof accomplishes all-program verification for early-stage verification. Meanwhile, RealityCheck and RTLCheck are the first methodologies and tools to provide automated intermediate-stage and post-implementation MCM verification for parallel processors, but they cannot verify across all programs. Building on RealityCheck and RTLCheck to develop methodologies capable of verifying MCM properties on detailed designs and RTL across all programs is an avenue of future work (Section 7.3.1). It should be possible to use RealityCheck and RTLCheck as a base for such all-program verification, given that PipeProof built on PipeCheck to do the same for early-stage designs.

A progressive verification flow need not have exactly three stages. Depending on how far removed early-stage design is from the eventual implementation of the system in question, engineers may choose to have more than one stage of intermediate verification between early-stage verification and post-implementation verification. On the other hand, if the system is simple enough that early-stage designs can be easily linked to low-level implementations, a distinct intermediate verification stage may not be necessary. The key idea is to conduct verification throughout the development timeline (as opposed to just post-implementation) and to link the verification methods at each stage together.

6.6 Chapter Summary

Validation of a system is crucial to ensure its correctness before it is released to the end user. In conventional development flows, validation generally takes place post-implementation and uses primarily testing-based methods. Beginning validation this late in development leads to design bugs and sometimes even high-level specification bugs not being discovered until after the design has been implemented. Furthermore, testing-based methods can easily miss bugs and are insufficient for ensuring system correctness.

In response, this chapter proposes a verification flow called *Progressive Automated Formal Verification*, which advocates for automated verification at multiple points in the development timeline, beginning with the verification of early-stage designs. Design-time verification detects design bugs and high-level specification bugs earlier than traditional practices, and can help eliminate certain classes of bugs before implementation commences. This reduces or eliminates the amount of time engineers spend creating implementations of incorrect designs, thus reducing overall development time.

Progressive verification also recommends linking the verification methods at different stages to each other. The linkage of early-stage verification methodologies to post-implementation verification can reduce overall verification overhead by verifying design properties at a higher level of abstraction. This linkage may be facilitated by the appropriate use of modelling and verification at an intermediate point in the development timeline. The formal nature of the verification in a progressive flow provides the strong correctness guarantees necessary for today's systems. The automation of the verification allows hardware and software engineers to formally verify their systems by themselves, without needing to rely on formal methods experts.

In addition to proposing the philosophy of progressive verification, this dissertation also enables the progressive verification of MCM properties in parallel processors. The work in Chapters 3, 4, and 5, when combined, forms a progressive verification flow for hardware MCM properties. The progressive verification flow itself is applicable to other domains beyond MCM verification, including hardware security verification and compiler verification.

Chapter 7

Retrospective, Future Directions, and Conclusion

"This isn't the end. There is no end."
—Hotta Yumi (translated)
Hikaru No Go

This chapter begins (Section 7.1) by situating the work in this dissertation in the context of the hardware/software stack. Specifically, all the work in this dissertation is part of the "Check suite", a set of methodologies and tools for MCM verification from high-level languages down to RTL. Section 7.2 then examines lessons learned retrospectively from doing the research presented in this dissertation. Section 7.3 then outlines avenues of future work that can build on the advances in this dissertation. The chapter ends with the conclusions of the dissertation (Section 7.4).

7.1 Zooming Out: The Check Suite for MCM Verification

The Check suite [LPM14, MLPM15, LSMB16, TML+17, MLMP17, MLMG18, MLM20] comprises a set of methodologies and tools developed at Princeton for automated MCM

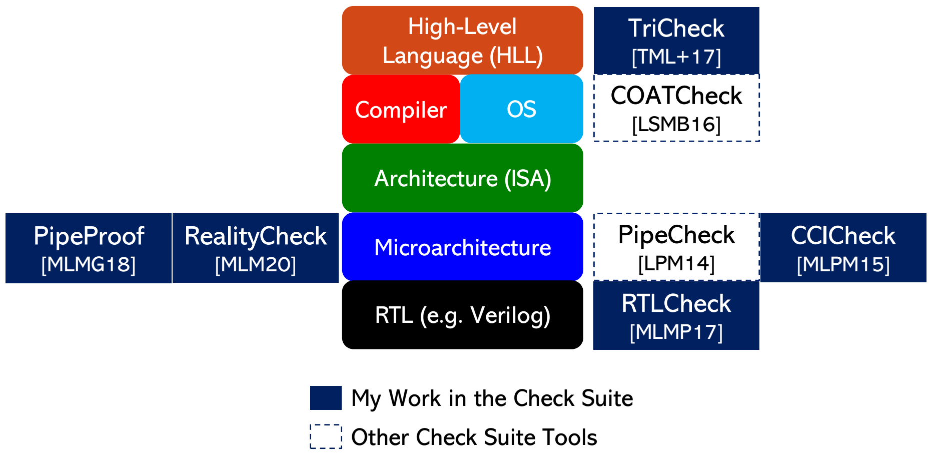


Figure 7.1: The Check suite of MCM verification tools developed at Princeton over the past 7 years. Each tool is positioned based on the layers of the hardware/software stack that it operates at. All of my work that is presented in this dissertation is part of the Check suite.

verification across the hardware/software stack, from high-level languages (HLLs) to RTL. Figure 7.1 depicts all the tools in the Check suite, shown according to the layers of the hardware/software stack they work at. PipeCheck [LPM14, LSMB16] (Section 2.4) was the initial tool in the Check suite. All later Check suite tools built on PipeCheck's initial modelling framework to accomplish their goals. The RTLCheck (Chapter 3), RealityCheck (Chapter 4), and PipeProof (Chapter 5) tools presented in this dissertation are part of the Check suite.

In addition to the work presented in this dissertation, I was also a co-author on the CCICheck [MLPM15] and TriCheck [TML+17, TML+18] tools from the Check suite. CCICheck and TriCheck are not discussed in detail in this dissertation due to length constraints. I provide a brief description of each tool below.

CCICheck conducted coherence[1]-aware microarchitectural MCM verification that was capable of modelling cache occupancy and coherence protocol events related to MCM verification. This allowed it to model architectures with nMCA coherence protocols as well as partially incoherent microarchitectures like some GPUs. CCICheck also enabled the modelling of coherence protocol features (e.g., the livelock-avoidance

[1]Section A.5 covers the basics of cache coherence and coherence protocols.

mechanism in the "Peekaboo" scenario [SHW11]) and their impact on MCM behaviour. CCICheck discovered a bug in the TSO-CC lazy coherence protocol that could result in an MCM violation [Elv15].

Meanwhile, TriCheck enabled users to formally verify hardware-software MCM compatibility for suites of litmus tests. As input, TriCheck takes a litmus test in a high-level language (HLL) like Java or C++, a compiler mapping[2], and a μspec microarchitecture specification. TriCheck automatically translates the HLL litmus test into its ISA-level equivalent using the provided compiler mapping, and then formally verifies whether the forbidden outcome of the HLL litmus test is observable on the microarchitecture specified by the μspec. TriCheck is useful both when designing the ISA-level MCM for a new architecture as well as for validating compiler mappings for a given HLL-ISA pair. TriCheck discovered numerous issues in the draft MCM specification of the open-source RISC-V ISA, and spurred the creation of the RISC-V memory model working group (of which I am a member) to develop a new formally specified MCM for RISC-V. A new MCM fixing the issues TriCheck discovered (and other issues) was ratified in September 2018 [RIS19].

TriCheck also discovered two counterexamples to compiler mappings from C/C++ atomics to the Power and ARMv7 ISAs. These mappings had supposedly been proven correct previously [BMO+12], but I isolated the flaw in the supposed proof that allowed the invalid mappings to slip through [MTL+16]. These findings by my co-authors and myself led to the discovery that the C/C++ memory model was unintentionally slightly stronger than necessary, precluding the use of efficient mappings for it for weaker architectures like Power and ARMv7. Lahav et al. [LVK+17] concurrently and independently discovered the same underlying issue. They also proposed a fix to this issue that has since been adopted by the standards committee.

[2]A compiler mapping is a translation scheme from high-level language synchronization primitives (e.g., C/C++ `atomic` operations) to assembly language instructions for a given ISA.

7.2 Lessons Learned: A Retrospective

The methodologies and tools developed by this dissertation all break new ground in automated hardware MCM verification. However, certain choices that I made when conducting this research made the development of my tools harder than it could have been. If I were to redo the research in this dissertation knowing what I know now, I would make two changes in my approach. Specifically, I would attach greater importance to supporting operational models in my work, and I would create a strong type system for the μspec language and its successors like μspec++ (Section 4.5). This section examines these lessons learned in detail, in the hope of making researchers aware of the issues involved. This knowledge will ideally help other researchers make better choices when conducting similar research in the future.

7.2.1 Importance of Operational Model Support

Section 2.2.3 describes two prominent modelling styles in formal MCM analysis: operational models and axiomatic models. Operational models describe a system being modelled as a transition system (Section 2.2.1), while axiomatic models describe a system using a set of invariants that hold in it.

PipeCheck chose to use axiomatic modelling for its μspec specifications, and provided no support for operational modelling of microarchitectures. Axiomatic models have notable benefits for MCM verification, namely concise specification and efficient verification [AMT14]. Thus, PipeCheck's choice of axiomatic modelling had significant advantages.

As seen in Chapters 3, 4, and 5, I chose to follow PipeCheck and only use axiomatic models for microarchitectural ordering specification and verification. However, this choice of purely axiomatic modelling has three disadvantages when compared to including support for operational models. Firstly, it makes it harder to link μspec models to RTL. Secondly, axiomatic models make it harder to model scenarios where a program changes over time (like self-modifying code). Thirdly, the invariant-based approach of axiomatic models makes it hard to write specifications for certain hardware behaviour. The rest of this section details the nature of these disadvantages.

Linking to RTL

Axiomatic models like μspec specifications are tricky to link to operational models like RTL. RTLCheck (Chapter 3) accomplished the linkage of μspec specifications to RTL and the soundness verification of μspec specifications by translating μspec axioms to test-specific SVA assertions. Part of the difficulty in translating μspec axioms to SVA lies in the fact that the logics and semantics of μspec and SVA are starkly different from each other. A set of μspec axioms forms an axiomatic model of microarchitecture, while SVA assertions are evaluated over an operational model of RTL [CDH+15].[3] If I had developed or used a method for specifying microarchitectural orderings operationally rather than axiomatically, those orderings would be easier to relate to RTL. Correspondences between microarchitectural states and RTL states could then be used to relate the two operational models for verification. This is the approach used by prior work, e.g., Burch and Dill [BD94] and ISA-Formal [RCD+16].

Modelling Program Change Over Time

Operational models are also generally better suited than axiomatic models to represent scenarios where the program being modelled changes over time. For instance, when self-modifying code executes, the program being run by the processor changes as it runs. However, in axiomatic frameworks like `herd` (Section 2.3.1), PipeCheck (Section 2.4), RealityCheck (Chapter 4) and PipeProof (Chapter 5), the entire program being run

[3]In an operational model of RTL, the values of the `wire`s and `reg`s in any given cycle constitute a state. A state s_1 can transition to a state s_2 if and only if the RTL allows the values of the `wire`s and `reg`s in s_1 to change to those of s_2 in one clock cycle.

(including the values returned by all loads) is known right at the outset and can be seen by all axioms. There is no notion of allowing the program instructions to change as the execution progresses. Axioms are evaluated on executions as a single unit.

On the other hand, in an operational framework, instructions can be executed piece-by-piece through state transitions, just like they are executed by a real processor's pipeline stages. Modification of the program is simply an update to the portion of the state that represents instruction memory, and can be conducted at an arbitrary point in an execution trace. If my work had provided a way to specify microarchitectural orderings operationally, it would have made it easier to model behaviour like self-modifying code.

Modelling Certain Hardware Structures

Axiomatic models describe systems in terms of invariants that the system upholds. These invariants are often straightforward to write for an entire system, but more difficult to write for certain parts of the system considered in isolation. For instance, consider the property of coherence. Most architectures implement per-location SC [AMT14, Int13, ARM13, IBM13, RIS19], which is slightly stronger than coherence. This property is straightforward to specify at the ISA level in `herd` for an entire processor as $acyclic(po\_loc \cup co \cup rf \cup fr)$. It is also straightforward to specify this property as a μspec axiom for an entire microarchitecture. Such an axiom would simply enforce an order between the memory hierarchy nodes or ViCLs[4] of any two writes to the same address.

However, what are the invariants for a *single* cache in a microarchitecture implementing a typical coherence protocol? The answer to this question is not so simple. These invariants would form the μspec++ axioms for the module representing that cache in a RealityCheck (Chapter 4) specification. The invariants do exist, and so

[4]An abstraction developed by CCICheck [MLPM15] to model cache occupancy and coherence events.

RealityCheck is indeed capable of modelling coherent caches. Even so, the difficulty in coming up with such invariants makes it somewhat difficult to model such caches in an axiomatic framework like RealityCheck.

On the other hand, specifying the behaviour of a single cache in a coherence protocol operationally is much more straightforward. Coherence protocols are routinely specified using state transition tables for each component in the protocol [SHW11], e.g., caches, main memory, directories. Such a transition table represents the states and transition relation of an operational model, making the construction of an operational model for such a protocol easy to do. If I had developed RealityCheck to allow certain modules to be specified operationally, it would be easier to specify hardware modules such as coherent caches.

Overall, both axiomatic and operational models each have their own advantages and disadvantages. Thus, it is desirable to have a single modelling framework that supports both operational and axiomatic models, as well as the ability to automatically convert between the two. Section 7.3.2 provides further details on this avenue of future work.

7.2.2 Benefits of a Type System for μspec and μspec++

PipeCheck developed the μspec domain-specific language to model microarchitectural orderings. μspec does not have a formally specified type system, and I did not create one for the language (or its successor μspec++) in the course of my dissertation research. A type system for a programming language can have several benefits [Pie02]. It makes it easy to detect certain classes of bugs, such as when the wrong number of arguments are provided to a function, or when one of the function's arguments has an incorrect type. Typing can also help ensure that a program does not get "stuck", i.e., reach a state where it is impossible to continue evaluation.

In μspec, if a user uses a predicate that does not exist or supplies an incorrect number of arguments to a predicate, PipeCheck and the tools in this dissertation do report that the predicate could not be found. However, these tools do not provide further detailed information about the error. A formal type system for μspec would facilitate providing such detailed error information. Users could be notified of both the type of the predicate they were trying to use and the expected type for that position in the specification, e.g., "`Line 24: Predicate 'IsAnyWrite' has 2 arguments, 1 expected`". Such information is readily available in compile errors for languages with rich type systems like Gallina.[5]

The value of a type system to a language for specifying microarchitectural orderings increases as the language becomes richer and incorporates more features. Such advancements to μspec were indeed added by the work in this dissertation, most prominently in Chapter 4. The μspec++ domain-specific language (Section 4.5) that I developed as part of RealityCheck adds support for modularity, hierarchy, and abstraction to μspec.

In RealityCheck, modules have internal and external nodes, corresponding roughly to events on internal and input/output signals in Verilog respectively. As Section 4.5.2 covers, a module should trigger certain events (external nodes) in another module but not others. For instance, the implementation axiom of the `Mem` module in Figure 4.10 does not assume that a request (`Req`) node exists for a given transaction. The request must be triggered by another module, like the `Core`. Similarly, the `Core` cannot assume that the response to its request will arrive; the response's existence must be enforced by axioms in the `Mem` module.

Ideally, one would want to annotate external nodes as either being `input` or `output` with respect to a given module. A module would only be able to enforce the existence of its own `output` events and the `input` events of other modules, similar

[5]Gallina is the functional programming language of the Coq proof assistant. The majority of the Check suite is written in Gallina.

to how input/output signals are treated in Verilog. Capturing the specific patterns in μspec++ that enforce the existence of certain nodes is somewhat tricky due to μspec++ not being a general-purpose programming language. However, annotations of node input/output attributes and reasoning about them could be encoded in a type system. A type check of the resulting μspec++ specification would then be able to flag any instances where a module was incorrectly enforcing the existence of certain nodes, e.g., if Figure 4.10 was enforcing the existence of the `Req` node for a transaction. Thus, the type system would effectively eliminate this class of bugs.

Given the advantages of type systems for general-purpose programming languages, there are likely other advantages to having a strong type system for μspec, μspec++, and other languages that build on them. These advantages stand to increase as hardware ordering specifications become richer and capable of describing new hardware features, such as those in emerging accelerator-rich architectures (Section 7.3.4).

7.3 Future Work

The advances in this dissertation set up a number of exciting lines of future work. This section details these future work ideas in broad strokes.

7.3.1 Furthering Automated All-Program MCM Verification

A multicore processor must obey its MCM for any program that it runs. As such, to truly ensure MCM correctness in all possible scenarios, hardware MCM verification must be complete, i.e., cover all possible programs. PipeProof (Chapter 5) demonstrates how to conduct automated all-program microarchitectural MCM verification. However, MCM bugs may be introducted later in development after PipeProof's early-stage verification has been conducted. Thus, PipeProof alone cannot ensure the MCM correctness of a taped-out chip across all programs.

As Chapter 6 covers, RealityCheck (Chapter 4) and RTLCheck (Chapter 3) are intended for MCM verification after PipeProof, i.e., later in the development timeline. It is their responsibility to ensure that MCM bugs do not enter the design or implementation after it has been verified with PipeProof. While RealityCheck and RTLCheck break new ground in scalable MCM verification and the MCM verification of RTL, they only verify litmus tests or executions up to a bounded length, and do not verify across all programs. Thus, while their verification is extensive, RealityCheck and RTLCheck leave open the possibility for MCM bugs to enter development after PipeProof's verification is conducted. As an example, a user may verify a microarchitecture's μspec specification using PipeProof, and then verify that RTL satisfies the microarchitecture's μspec axioms for a suite of litmus tests using RTLCheck. Even if they do so, it is possible (though unlikely) that the RTL does not satisfy the μspec axioms for a program that is not in the litmus test suite. If this is the case, then the RTL contains an MCM implementation bug and is incorrect.

To ensure all-program correctness of a taped-out chip, all MCM verification methodologies used in the development timeline must be complete. PipeProof built on the litmus test-based verification of PipeCheck (Section 2.4) to develop its all-program verification methodology. Similarly, future work can build on RealityCheck and RTLCheck to develop automated all-program MCM verification methodologies for detailed designs and RTL respectively. When combined with PipeProof, the resultant set of three tools would constitute a progressive verification flow (Section 6.5) for MCM properties that ensures correctness across all programs. This will ensure that from early-stage design through to the end of RTL implementation, no MCM bugs exist in the processor.

The need for all-program verification also exists for software that is part of MCM implementations, like compilers. A compiler mapping must be verified to ensure that any HLL program compiled with it will maintain the guarantees of the HLL MCM (provided that the hardware is correct). TriCheck [TML+17] provided early-stage MCM verification of compiler mappings across litmus tests using an approach based on PipeCheck and `herd` [AMT14]. Future work can build on TriCheck to develop an automated all-program MCM verification methodology for compiler mappings, as PipeProof did with PipeCheck.

7.3.2 A Unified Modelling Framework Supporting Axiomatic and Operational Models

As Section 2.2.3 describes, operational and axiomatic models each have their own advantages and disadvantages. Axiomatic models tend to enable concise system specifications and more efficient verification methodologies than those for operational models. On the other hand, users may find it easier to specify a system using an operational model, since doing so does not require knowledge of the invariants of the system. Operational models may also be more intuitive because they tend to resemble the systems that they model, especially in the case of hardware.

The axiomatic modelling used by the Check suite enables the concise specifications and efficient verification methodologies of its tools, including RealityCheck (Chapter 4) and PipeProof (Chapter 5). However, as Section 7.2.1 describes, the use of operational modelling would have made certain verification tasks easier, such as linking to RTL. Ideally, one would have two specifications for a given system, one axiomatic and one operational, and the two would be proven to be equivalent to each other. Each specification could then be used in the scenarios where it would be most beneficial. For instance, if modelling microarchitectural orderings, one would use the axiomatic specification for microarchitectural MCM verification but the operational version for linking to RTL.

Currently, to get the benefits of both operational and axiomatic models, formal methods researchers create both types of specifications and then manually prove them equivalent to each other [OSS09, MHMS+12, AMT14, PFD+18]. A more optimal approach would be to have a method to automatically generate an equivalent operational model for a given axiomatic model (and vice versa). The insights I gleaned from developing RTLCheck's procedure for translating μspec axioms to SVA assertions (which run on an operational model) may prove useful to enable this conversion. In addition, for the case of generating an axiomatic model from an operational specification, invariant synthesis techniques like I4 [MGJ+19] may be beneficial.

Another issue covered in Section 7.2.1 is that certain systems (or parts of systems) are easier to specify axiomatically, while others are easier to specify operationally. Ideally, one would be able to specify components of a system in the manner most intuitive to that component, and then connect the various components together to generate the overall system specification. The ILA-MCM [ZTM+18] framework that I was part of accomplished connection between an operational model of instruction semantics and an axiomatic concurrency model based on μspec axioms. Future work can build on this research to develop a generic framework capable of supporting and connecting both operational and axiomatic models.

To summarise, an important line of future work in formal modelling is to develop or modify a modelling framework to support (i) the specification of both operational and axiomatic models, (ii) the straightforward connection of operational and axiomatic components of a model to each other, and (iii) the automatic generation of an equivalent operational model for a given axiomatic model (and vice versa).

7.3.3 Progressive Verification of Other Domains

The progressive verification flow (Figure 6.2) is not specific to the domain of MCM verification. This flow can also be used for other types of properties. Progressive verification is most useful for properties which can be broken by bad designs as well as by bad implementations. Hardware security properties fit this description, and so a progressive flow can be used to verify them. CheckMate [TLM18a] provides bounded early-stage hardware security verification based on the μhb graphs developed by PipeCheck. A methodology similar to RealityCheck could provide such verification for detailed designs, while a tool like RTLCheck could provide post-implementation verification of the relevant security properties. Together, the three tools would form a progressive verification flow, similar to what PipeProof, RealityCheck, and RTLCheck do for MCM properties.

Progressive verification is not limited to hardware verification either. For example, a compiler could be verified using a progressive flow. In such a flow, early-stage verification of the compiler would verify its basic code generation scheme. For instance, TriCheck [TML$^+$17] provides early-stage MCM verification of compiler mappings as one of its features. Verification later in the design process could introduce more advanced compiler functionality, such as optimizations (e.g., Dodds et al. [DBG18]). Finally, post-implementation verification could verify the actual compiler code itself using automated approaches (e.g., akin to CompCert [Ler09], but automated).

7.3.4 Developing Abstractions and Concurrency Models for Emerging Hardware

In 1964, Amdahl et al. [ABB64] described the "architecture" of the IBM 360 as the programmer-facing attributes of its hardware, like its functional behaviour. This "architecture" subsequently evolved into the hardware/software ISA abstraction we know today. For decades, the ISA defined the hardware/software interface and served as a contract between hardware and software. Software targeted the ISA, and hardware implemented and was verified against the ISA. However, as this section covers, the ISA has now become a barrier to continued hardware improvement.

The end of Moore's Law and Dennard scaling has led to the rise of today's heterogeneous parallel architectures (Section 1.1), where designs are replete with accelerators. Hill and Reddi [HR19] define *accelerator-level parallelism* (ALP) as "the parallelism of workload components concurrently executing on multiple accelerators". As an example of ALP, they show that 4K video capture on a smartphone may use multiple accelerators in a parallel and pipelined fashion, including an image signal processor, a GPU, and a DSP (digital signal processor). In today's SoC designs, especially those of smartphones, ALP is becoming the norm. As an example, the recent Apple A12 SoC has more than 40 accelerators [WS19]. Each of these accelerators is meant for a different purpose, and they are sometimes programmed using different toolflows. For instance, TensorFlow [Goo20] and PyTorch [F$^+$20] are sometimes used to program machine learning accelerators.

The legacy ISA hardware/software abstraction is geared towards specifying the behaviour of a general-purpose processor, not an SoC like today's smartphone chips. ISAs generally cannot specify the behaviour of accelerators, especially since some accelerators may not even have instruction sets [SVRM15]. Computing is moving towards a "post-ISA" world [Mar18], where toolflows routinely bypass the ISA to program accelerators directly. Thus, the ISA is proving insufficient for modelling emerging heterogeneous parallel hardware.

These developments necessitate the creation of new hardware/software abstractions capable of describing the behaviour of emerging hardware. Without such specifications, achieving system correctness becomes difficult. In the absence of a hardware/software interface specification, software does not have a clearly defined interface to target, and hardware does not have a specification to be validated against.

The hardware/software abstractions of the future should be formally specified so that they unambiguously describe the required software-facing behaviour of the hardware. An example in this vein is the ISA-Formal project [RCD$^+$16] from ARM, which mechanised the entire ARM instruction set specification so as to facilitate verification (among other things). While a staggering achievement, ISA-Formal does not support the modelling of accelerators. The RISC-V community is also attempting to formalise their instruction set specification. They have flagged ease of extension to plug in formal specifications of accelerators and I/O devices as a desirable trait [Nik19], so as to better model heterogeneous parallel SoCs.

A promising line of research in the development of new hardware/software abstractions is the Instruction-Level Abstraction (ILA) [SVRM15]. ILAs are ISA-like abstractions of accelerators, and can be automatically generated from an implementation and an ILA specification template. ILAs can thus serve as targets for accelerator software/firmware as well as specifications for hardware to be verified against. In an SoC with several accelerators like the Apple A12, each accelerator can be described with an ILA.

When multiple accelerators communicate with each other on an SoC, they often do so via shared memory, as in Hill and Reddi's 4K video capture example referenced above [HR19]. ALP in SoCs thus necessitates the development of MCM specifications describing the SoC's shared memory behaviour. These MCMs must also be formally specified to avoid ambiguity and facilitate verification (Section 2.1.4). During my PhD, I was part of the ILA-MCM [ZTM$^+$18] follow-on work to the initial ILA paper. This work linked ILA models of instruction operational semantics with axiomatic MCM constraints reflecting the nature of shared memory in order to facilitate SoC verification. Future work can build on ILA-MCM's example to develop rich formal models for emerging heterogeneous parallel hardware. These models should include both instruction operational semantics and an MCM.

MCM specifications for emerging hardware should be driven by the nature of the hardware involved, specifically its memory access patterns and reordering behaviour. RealityCheck's support for modularity and abstraction is well-suited to modelling and abstracting the behaviour of such heterogeneous parallel hardware, and can inform the development of such MCMs. Users of RealityCheck can specify the ordering properties of each processing element (including accelerators) on an SoC individually and then compose them together to create a detailed ordering specification of the overall SoC. They can then judiciously use interfaces and interface verification (Section 4.4) to soundly (up to a bound) abstract the ordering behaviour of the SoC step-by-step to generate an ordering specification that is closer in granularity to today's ISA-level MCMs.

Once formal MCM specifications for emerging hardware are created, the tools developed by this dissertation (PipeProof, RealityCheck, and RTLCheck) can be used to verify designs and implementations against these specifications. RealityCheck in particular is well-placed to verify heterogeneous parallel architectures thanks to its support for modularity, hierarchy, and abstraction.

7.4 Dissertation Conclusions

Computers are becoming more and more integrated into our daily lives. Today, we routinely use computers to hold teleconferences, navigate using GPS, work remotely, and for a plethora of other tasks. With the coronavirus pandemic and the rise of AI and machine learning, the importance of computing in our daily lives will continue to increase.

To be capable of accomplishing all these tasks, computers have become more complex over the years, and their complexity continues to increase. In the hardware world, the end of Moore's Law and Dennard scaling has resulted in the emergence of heterogeneous parallelism in recent years. SoCs today routinely have at least 4 to 8 general-purpose processing cores that can operate in parallel, and they may have over 40 accelerators. Each accelerator is designed for a specific type of computation, and may have its own programming toolchain. Processor components are developed by a number of distinct teams, and at the SoC level, components may even be developed by different vendors.

Complex systems like those of today are more bug-prone than simple ones. At the same time, the importance of computing to our daily lives means that the ramifications of these bugs are higher than they have ever been. This necessitates stringent verification to ensure the correctness of today's processors and SoCs. Hardware verification is a general challenge, with verification costs now dominating total hardware design cost [Fos15].

Despite this investment, hardware bugs continue to slip through industry verification practices into chips that are released to end users. While software bugs can generally be easily patched, hardware typically cannot be modified after its release to the end user. Fixes for hardware bugs therefore often employ heavy-handed workarounds that notably decrease processor capabilities. For instance, the recent Spectre hardware security bug had to be fixed using fence instructions that reduced performance [KHF$^+$19]. Similarly, a bug in Intel's transactional memory implementation on Haswell processors was fixed by disabling the transactional memory functionality [Hac14]. Thus, it is critical that hardware bugs be discovered as early as possible.

Part of the problem with hardware verification today is that processor manufacturers still use primarily testing-based methods for validation. These methods can easily miss bugs that occur only in very specific circumstances, especially on today's nondeterministic multiprocessors. Verification using formal methods, on the other hand, is adept at discovering hard-to-find bugs and can provide strong correctness guarantees for designs and implementations. However, the use of formal verification often requires deep expertise in formal methods, which typical hardware engineers do not have.

In today's parallel architectures, processing elements often communicate and synchronize with each other through loads and stores to shared memory. Memory consistency models (MCMs) specify ordering rules for load and store operations to shared memory, thus constraining the values that can be read by load operations. A processor must respect the MCM of its ISA for any program that it runs, or the correct operation of a parallel program on that processor cannot be guaranteed. Verification of MCM implementations is thus critical to overall parallel system correctness. Furthermore, emerging heterogeneous parallel designs are likely to employ accelerator-level parallelism [HR19] (multiple accelerators concurrently working together to process a workload on chip). These accelerators can also communicate with each other via shared memory, so MCM verification will continue to be important for emerging designs.

This dissertation makes a number of novel contributions that significantly improve the state of the art in parallel hardware verification, in particular that of MCM properties. The work in this dissertation develops MCM verification methodologies and tools that can deliver the standard of verification required for real-world commercial hardware. The automated tools are designed to be used by hardware engineers, enabling them to obtain strong correctness guarantees about their designs and implementations by themselves. The three tools (PipeProof, RealityCheck, and RTLCheck) can be used together in the flow of *Progressive Automated Formal Verification* for thorough MCM verification of hardware designs and implementations throughout their development. Progressive verification can detect bugs earlier and reduce overall development time, thus helping to address the large amount of time spent on hardware verification today.

Individually, the tools in this dissertation provide verification coverage across all programs (PipeProof), verification that is scalable (RealityCheck), and verification of real implementations (RTLCheck). All-program verification coverage ensures that no bugs slip through the verification, while scalable verification ensures that the methodology is capable of tackling the increasing complexity of hardware today. Meanwhile, linking verification of formal models to that of real implementations helps push correctness guarantees for early-stage designs through to chips that are shipped to end users.

Overall, the contributions of this dissertation are as follows:

- Chapter 3 presents RTLCheck, a methodology and automated tool for linking microarchitectural ordering specifications to RTL implementations. Prior work on automated hardware MCM verification only went down to microarchitecture and could not be linked to real implementations. RTLCheck enables such linkage, allowing correctness guarantees proven for early-stage designs to be easily pushed down to the eventual RTL implementations. RTLCheck doubles as a mechanism to formally verify the soundness of a microarchitectural model with respect to RTL, thus helping users develop accurate formal models of existing processor implementations.

- Chapter 4 presents RealityCheck, a methodology and automated tool for scalable microarchitectural MCM verification of detailed designs. Prior work on automated microarchitectural MCM verification used monolithic approaches that do not scale due to the NP-completeness of the SMT solvers used. RealityCheck exploits the structural modularity inherent in hardware designs to enable specification and verification of a design piece-by-piece. This allows for scalable verification by breaking up a processor's MCM verification into smaller verification problems. RealityCheck's modular specifications are also an excellent fit for the distributed nature of the hardware design process.

- Chapter 5 presents PipeProof, a methodology and automated tool for all-program microarchitectural MCM verification of early-stage designs. Processors must be verified as correctly respecting their MCMs across all possible programs to ensure correctness. However, prior automated microarchitectural MCM verification approaches only provided bounded verification, which only guaranteed design correctness for a subset of all programs. PipeProof develops the first microarchitectural MCM verification approach capable of automatically verifying MCM correctness across all programs, giving designers complete confidence that there are no MCM bugs in their design.

- Chapter 6 presents *Progressive Automated Formal Verification*, a novel generic verification flow with multiple benefits. Prior formal verification approaches focused on one point in development, like early-stage design or post-implementation verification. In contrast, progressive verification emphasises the use of automated formal verification at multiple points in the development timeline and the linkage of the different verification approaches to each other. Progressive verification enables the earlier detection of bugs and provides reductions in verification overhead and overall development time. The combination of PipeProof (Chapter 5) with RealityCheck (Chapter 4) and RTLCheck (Chapter 3) enable the progressive verification of MCM properties in parallel processors. This concrete instance of a progressive verification flow serves as a reference point for future work on the progressive verification of other types of properties and systems.

- Overall, the work in this dissertation advances automated formal MCM verification much closer to being capable of verifying the designs and implementations of real-world processors. Individually, each of the tools in this dissertation makes its own contribution in this regard. RTLCheck enables automated MCM verification of real processor implementations for the first time. RealityCheck's twin benefits of scalability and distributed specification are both critical to the verification of real-world designs, while PipeProof brings the coverage of automated MCM verification approaches up to the level required for real-world processors. In addition, when the the three tools are combined in a progressive verification flow, they enable thorough and efficient MCM verification across much of the hardware development timeline. This thorough progressive verification is essential to ensure the MCM correctness of real-world processors that are shipped to end users.

This dissertation addresses major outstanding challenges in hardware MCM verification, and the methodologies and tools I present here are well-positioned to handle the hardware architectures of the future. However, MCMs are but one of the types of properties that must be verified for the hardware designs of today and the future. Processors must also be verified as being functionally correct with respect to the semantics of their individual instructions (or instruction-level operations, in the case of accelerators). Proving the functional correctness of future processors will be harder than it has been in the past, as each type of accelerator may have its own correctness criteria.

Given how much of our lives we live online today, the processors of the future must also be secure. The recent Spectre [KHF$^+$19] and Meltdown [LSG$^+$18] attacks have brought to light a class of serious security issues with today's designs. Future processors must protect against these and other hardware security issues, so that the use of tomorrow's hardware does not come with risks of information leakage and identity theft. Furthermore, with the rise of AI and machine learning, questions of ethics and bias in computing results have also become more prominent. The capability to evaluate AI and machine learning systems for bias is critical to ensuring that the benefits of AI and machine learning are equally available to all.

Verification using formal methods can provide the strong correctness guarantees required by the hardware and software of tomorrow. However, the use of formal methods has historically been restricted to the relatively small set of individuals with formal methods knowledge. The formal verification tools of the future must be usable by typical hardware and software engineers, so as to enable formal verification to become mainstream. The verification tools developed in this dissertation have usability as one of their core tenets, and show that improving usability need not result in a reduction in verification capabilities. The inclusion of basic formal methods instruction in computer science and engineering curricula would also make it easier for the engineers of the future to utilise formal verification tools.

The progressive verification flow proposed in this dissertation advocates for stringent yet efficient verification of computing systems throughout their development. It is a generic verification flow, and should be applied to other types of properties and systems in the future. Progressive verification of such properties and systems will bring the same rigour to their verification that this dissertation brings to hardware MCM verification. The computing community as a whole must work towards a future where bugs are rarities rather than the norm, and where formal verification tools are a standard part of every hardware and software engineer's toolkit. The importance of computing to our lives today makes it paramount that we do so.

Appendix A

Hardware Features and Attributes That Impact MCM Behaviour

This appendix briefly covers some hardware features and attributes that affect a processor's MCM behaviour, including out-of-order execution, write atomicity and cache coherence, dependencies, and cumulativity. The material in this appendix draws on the existing literature, including Adve and Gharachorloo's MCM tutorial [AG96], the primer on consistency and coherence by Sorin et al. [SHW11], and Hennessy and Patterson's textbook on computer architecture [HP17]. The intent is to give the reader a feel for how common hardware features give rise to the counterintuitive weak behaviours and complicated specifications seen in many of today's MCMs.

A.1 Non-FIFO Coalescing Store Buffers

Store buffers can cause the reordering of write instructions with subsequent reads (Section 1.2). The most restrictive store buffers are FIFO, which means that they do not reorder stores with respect to each other. However, this can result in a scenario where stores that could go to the cache get blocked behind a store that is waiting for its cache line to be fetched from memory (assuming a write-allocate cache). FIFO

Core 0	**Core 1**
(i1) [x] ← 1	(i3) r1 ← [y]
(i2) [y] ← 1	(i4) r2 ← [x]
SC forbids r1=1, r2=0	

Figure A.1: Code for litmus test `mp`

Core 0	**Core 1**
(i1) [y] ← 1	(i4) r1 ← [y]
(i2) [x] ← 1	(i5) r2 ← [x]
(i3) [y] ← 2	
SC forbids r1=2, r2=0	

Figure A.2: Code for litmus test `mp+w`

store buffers also prevent the coalescing of two store buffer entries with the same address into a single entry, as doing so can reorder stores with respect to each other (or loads with subsequent stores). In order to allow non-FIFO coalescing store buffers, either the processor's MCM must allow Store-Store reordering, or the processor must be capable of speculatively reordering the stores (Section 2.1.2).

For instance, if running `mp` (Figure A.1) on a processor whose cores have non-FIFO store buffers, then even if the cores have in-order pipelines, the non-SC outcome of `mp` will be possible. The store buffer may reorder the two stores `i1` and `i2` (even if they were sent to the store buffer in order), causing them to reach memory out of order. This reordering can then be observed by core 1, giving the outcome `r1=1,r2=0`.

For an example of write coalescing, consider Figure A.2's litmus test `mp+w`. This test is the same as `mp` except for two differences: the store and load of `y` have a value of 2 rather than 1, and there is an additional store of 1 to `y` on core 0 (instruction `i1`) preceding the other instructions on that core. Assume that core 0 sends its three store instructions to its store buffer in order. If running on a processor with coalescing store buffers, core 0's store buffer may merge its entry for `i3` into the existing entry for `y` that was previously created for `i1`. However, this effectively reorders instructions `i2` and `i3`, enabling the outcome `r1=2,r2=0` which is forbidden by SC.

Core 0	Core 1
(i1) r1 ← [x]	(i3) r2 ← [y]
(i2) [y] ← 1	(i4) [x] ← 1
SC forbids r1=1, r2=1	

Figure A.3: Code for litmus test `lb`

A.2 Out-of-order Execution

In a single-core system, the out-of-order execution of memory operations can provide significant performance benefits, but can result in the reordering of memory-accessing instructions [AG96, HP17]. In addition to the reorderings already mentioned, it also enables the reordering of load instructions with subsequent loads or subsequent stores. This optimization does not change architectural behaviour in a single-core system, as no other core exists to observe the reordering. For instance, in the `mp` litmus test (Figure A.1), if core 1's instructions are considered in isolation, the loads can be performed out of order with no changes in the values they return (since in a single-core system no other core can write to those addresses while the loads are executing). However, in a multicore system, the reordering of `i3` and `i4` can result in the non-SC outcome of `r1=1,r2=0` becoming observable, even if the stores `i1` and `i2` are performed in order.

For an example of Load-Store reordering, consider the `lb` (load buffering) litmus test (Figure A.3). This litmus test has two addresses, `x` and `y`. Each core has a read to one of these addresses followed in program order by a write whose address is read by the other core. The outcome `r1=1,r2=1` is forbidden under SC, and requires at least one pair of instructions (i.e., `i1` and `i2` or `i3` and `i4`) to be reordered with each other to occur. Out-of-order execution enables such reorderings and can thus result in `lb` being observable on hardware.

```
lwz r1,0(r2);
xor r3,r1,r1;
lwzx r4,r3,r5;
```

(a) Load to load address dependency on Power.

```
lwz r1,0(r2);
xor r3,r1,r1;
addi r3,r3,1;
stw r3,0(r4);
```

(b) Load to store data dependency on Power.

```
lwz r1,0(r2)
cmpw r1,r1
beq  LC00
LC00:
lwz r3,0(r4)
```

(c) Load to load control dependency on Power.

Figure A.4: Examples of address, data, and control dependencies using Power assembly code.

The reordering of loads with subsequent load and store operations in program order brings up the question of whether dependent instructions will be reordered with each other. Section A.3 discusses dependencies in detail.

A.3 Dependencies

Many MCMs today (e.g., those of Power, ARMv7, ARMv8, and RISC-V) allow loads to be reordered with respect to subsequent loads and stores in program order. This raises the question of whether loads can be reordered with subsequent loads or stores which are dependent on them. Architectures like Power, ARMv7, ARMv8, and RISC-V define three types of dependencies: address, data, and control [SSA+11]. Figure A.4 shows examples of each type of dependency [AMT14]. An *address* dependency occurs when the address accessed by a load or store depends on the value returned by a load preceding it in program order. For example, in Figure A.4a, the second load is dependent on the first load. A *data* dependency exists between a load and a subsequent store when the store's value depends on the loaded value. This is the case in Figure A.4b. A *control* dependency occurs when the control flow decision of whether to execute a load or store depends on the value returned by a preceding load in program order. For example, in Figure A.4c, the result of the first load dictates

Core 0	**Core 1**
(i1) [x] ← 1 (i2) r1 ← [x] (i3) r2 ← [y]	(i4) [y] ← 1 (i5) r3 ← [y] (i6) r4 ← [x]
SC forbids r1=1, r2=0, r3=1, r4=0	

Figure A.5: Code for litmus test `iwp2.4`

Core 0	**Core 1**	**Core 2**	**Core 3**
(i1) [x] ← 1	(i2) [y] ← 1	(i3) r1 ← [x] (i4) r2 ← [y]	(i5) r3 ← [y] (i6) r4 ← [x]
SC forbids r1=1, r2=0, r3=1, r4=0			

Figure A.6: Code for litmus test `iriw`

whether or not the branch is taken, and the second load is after the branch in program order.

Power, ARMv7, ARMv8, and RISC-V all respect address and data dependencies, as well as control dependencies from loads to stores [SSA+11, PFD+18, RIS19]. Power, ARMv7, and ARMv8 enforce control dependencies from loads to subsequent loads if there exists an `isync` (on Power; the ARM equivalent is `isb`) after the branch that enforces the control dependency and before the second load (i.e., the dependee). Dependencies enforce local ordering and can thus be used as a lightweight ordering enforcement mechanism [SSA+11]. However, they do not enforce cumulative ordering (Section A.6), i.e., they do not order accesses on other threads.

Intuitively, it may seem impossible not to enforce dependencies, as a dependee seemingly cannot execute until it has all of its inputs available. However, in the presence of value speculation (i.e., speculatively predicting the values of loads), the dependee can in fact behave as if it were reordered with the instruction it depends on [MSC+01].

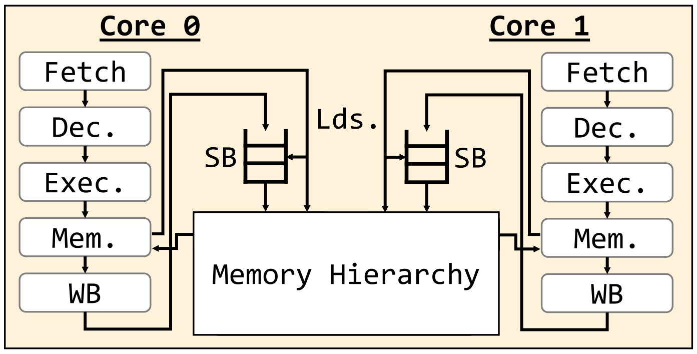


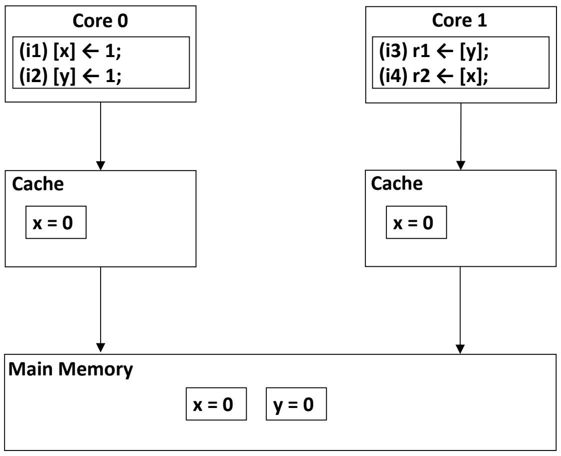
Figure A.7: A microarchitecture with two cores, each with five-stage pipelines and a store buffer (SB). This design is a simplistic example of how common hardware optimizations like store buffers can lead to weak MCM behaviours.

A.4 Relaxing Write Atomicity

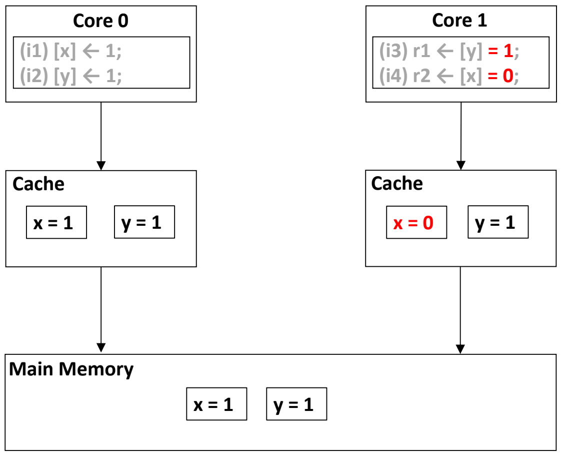
Values written by stores to memory do not just exist in main memory. They are also present in other locations as instructions execute, including the pipeline, store buffer, and caches. Each core only has access to a subset of these locations. For instance, each core can only read from its own pipeline and store buffer, and not those of other cores. Caches may also be private to each core. Thus, when a value is written to one of these locations, it may be visible to one core but not to others. *Write atomicity* concerns the notion of whether or not a store instruction is observed by all cores at the same time. Write atomicity adds another dimension to weak memory behaviours beyond the relaxations of program order by pipelines discussed previously.

If a store becomes visible to all cores in the system at the same time, the system is said to be "multi-copy-atomic" [Col92] (referred to as MCA in this dissertation). Multi-copy-atomicity is at odds with a number of microarchitectural features. Consider a microarchitecture with per-core store buffers (Figure A.7). When a store to an address `x` is put in the store buffer and is followed in program order by a load to `x`, the load must read its value from the store buffer to ensure single-thread correctness. However, this can result in the store becoming architecturally visible to core 0 before all other cores. Consider the `iwp2.4` litmus test in Figure A.5. This is a variant of `sb` where each core reads the address that they stored to before reading the value of the other core's flag. The outcome `r1=1,r2=0,r3=1,r4=0` corresponds to an execution where core 0 observes the store to `x` before the store to `y`, while core 1 observes the store to `y` before the store to `x`. If each store was becoming visible to all cores at the same time, this would be impossible. This behaviour, where a store becomes visible to the writing core before it becomes visible to all other cores is known as "reading your own write early" (referred to as rMCA in this dissertation). Note that under rMCA, when the store becomes visible to a core other than the writing core, it becomes visible to all other cores in the system at that same time. In Figure A.7's microarchitecture, this is intuitively because the store becomes visible to other cores by leaving the store buffer and going to memory (assume a single unified memory for now), which is observable by all cores.

A further relaxation of write atomicity, called "non-multi-copy-atomicity" (referred to as nMCA in this dissertation) is also possible. Under this variant, a store can become visible to cores other than the writing core at different times. Consider the `iriw` (Independent Reads Independent Writes) litmus test from Figure A.6. This is essentially `iwp2.4` with each store on a separate core with no other instructions. Now the outcome `r1=1,r2=0,r3=1,r4=0` can only occur if the store `i1` becomes visible to core 2 before core 3, while the store `i2` becomes visible to core 3 before core 2. Such behaviour can occur in a system where store buffers are shared between cores. If cores 0 and 2 share a store buffer, and cores 1 and 3 also share a store buffer, then the forbidden outcome of `iriw` can arise as follows. First, core 0 places its store of `x` in the store buffer it shares with core 2, and core 1 places its store of `y` in the store buffer it shares with core 3. Then, cores 2 and 3 perform their loads in program order. Thus, shared store buffers result in an nMCA system.



(a) Start of mp execution.



(b) End of mp execution.

Figure A.8: An illustration of how incoherent caches can break SC using the mp litmus test. Even though all instructions are executed in program order and one at a time, the forbidden outcome still becomes observable.

On nMCA architectures, fences must be *cumulative* in order to enforce ordering strong enough to restore SC. Cumulative fences order accesses on cores other than the core executing the fence. Section A.6 discusses cumulativity in further detail.

So far, this section has assumed a unified memory. However, a processor's memory hierarchy is commonly a collection of caches and main memory, with each core having at least one private cache. The caches in these memory hierarchies must be kept in sync with each other to provide the illusion of shared memory to the programmer. Processors commonly keep caches in sync using a *cache coherence protocol*, which can implement MCA, rMCA, or nMCA depending on its design. Section A.5 discusses coherence protocols in detail.

A.5 Cache Coherence and its Relationship to MCMs

Caches are used in almost every processor today to reduce memory latency and thus improve performance. In a single-core processor, caches are architecturally invisible[1] and generally do not cause the program to return different results. However, in a multicore processor, it is common for each core to have its own cache. Updates to one cache must somehow be propagated to the caches of other cores in order to keep the caches in sync with each other to present a *coherent* view of memory. Caches in most multiprocessors are kept in sync with each other through a *cache coherence protocol*[2] [SHW11]. The coherence protocol ensures that messages are exchanged between caches to update each cache with the writes that other cores have made to their caches.

The operations of a coherence protocol are a critical component of MCM enforcement in most shared-memory multiprocessors. As an example, consider the execution of `mp` (Figure A.1) depicted in Figure A.8, where each core has a private cache. At the start of execution (Figure A.8a), each cache contains a line for address `x` that has a value of 0. If core 0 writes a value of 1 to `x` in its cache and main memory[3] by executing instruction `i1`, then in the absence of a coherence protocol, this write will not be propagated to core 1 until the cache line for `x` in core 1's cache is evicted. Without a mechanism to enforce coherence, core 0 has no way to force core 1 to see the new value of `x`. This is problematic because the system's MCM may require core 1 to observe the new value of `x` before fetching other new values into its cache. (This is often the case for strong MCMs like SC and TSO.) For instance, core 0 may then

[1]Here, I am restricting myself to considering the results of instructions rather than side channels like the time an instruction takes to execute. Attackers can exploit the presence of caches to conduct side-channel attacks to leak confidential information [KHF+19].

[2]Some systems like GPUs may have some of their caches be incoherent [MLPM15] due to the nature of their architecture and the applications that are run on them.

[3]Assuming a write-through cache [HP17].

perform i2 and write 1 to address y in its cache and main memory. Core 1 may then perform i3, fetching the line for y from main memory, which contains its updated value of 1. Finally, core 1 can read its cache line for x and return a value of 0 for i4. Thus, without the coherence protocol, the system violates SC despite each core performing its memory operations in program order and one at a time. The orderings enforced by coherence play an integral role in the enforcement of weaker MCMs as well; this relationship can be illustrated through other litmus tests [SSA+11].

A coherence protocol enforces two invariants: the Single-Writer Multiple-Readers invariant (SWMR) and the Data Value Invariant (DVI) [SHW11]. The SWMR invariant enforces that only one core has write permissions (or multiple cores have read permissions) to a given cache line at any time. The DVI invariant enforces that each load returns the value of the latest store to its address. Together, these two invariants enforce a total order on all stores to a single address, and ensure that load operations return the value of the latest store to that address. So for instance, if one core were to observe the value of an address z changing from $0 \rightarrow 3 \rightarrow 2$, then all other cores would also observe that same sequence of updates to z.

Coherence protocols can be divided into two major types: those that use update-based coherence and those that use invalidation-based coherence [SHW11]. Update-based coherence protocols send the updated value for an address to other cores when it is written. Meanwhile, invalidation-based coherence protocols invalidate the line containing the address written to by one cache in the caches of other cores. In today's multiprocessors, invalidation-based coherence is largely used [SHW11]. Coherence protocols can also be divided into snooping protocols (which use a common bus on which all cores observe memory transactions) and directory protocols (where a centralised or distributed directory keeps track of which cores have access to which data, and enforces coherence by sending appropriate messages) [SHW11]. A coherence protocol may implement one of the flavours of write atomicity (Section A.4), depending

Core 0	**Core 1**	**Core 2**
(i1) [x] ← 1	(i2) r1 ← [x] (i3) <fence> (i4) [y] ← 1	(i5) r2 ← [y] (i6) <fence> (i7) r3 ← [x]
Cumulative fence `i3` forbids r1=1, r2=1, r3=0		

Figure A.9: Code for litmus test `wrc+fences`

on the requirements of the processor's MCM. Invalidation-based coherence protocols that implement nMCA allow a cache to send invalidations to other caches and not wait for their acknowledgement [PFD+18].

Coherence protocols are a widely studied area of research within computer architecture, including their design [SHW11, CKS+11, ASA18], verification [ZLS10, ZBES14, BEH+17], and even automatic generation [ONS18]. While coherence and memory consistency are often considered conceptually separate, their implementations are often closely coupled to improve performance [MLPM15]. For instance, coherence protocols can take advantage of the reorderings permitted by weak MCMs to reduce their bandwidth usage [KCZ92]. CCICheck [MLPM15] explored the relationship between coherence and consistency in multiprocessors, defining the *coherence-consistency interface* or CCI as the orderings that the coherence protocol provides to the rest of the microarchitecture combined with the orderings that the rest of the microarchitecture expects from the coherence protocol. The coherence protocol and the rest of the microarchitecture must agree on the distribution of responsibilities for MCM enforcement, or MCM violations will result.

A.6 Cumulativity

On nMCA architectures like Power and ARMv7, fences need to be *cumulative* to enforce ordering strong enough to restore SC. In addition to ordering memory operations before and after them on the same core, cumulative fences also order accesses on other threads observed by their thread before the fence before those that their thread observes after the fence. Consider Figure A.9's litmus test `wrc+fences` (`wrc` stands for Write-to-Read Causality). In this test, the fences enforce ordering between `i2` and `i4` as well as between `i5` and `i7`. Core 1 observes (`i2`) core 0's write to `x` (`i1`), and then writes 1 to `y` (`i4`). Core 2 observes (`i5`) core 1's write to `y`, and then reads the value of `x` (`i7`). Intuitively, one would expect that due to causality, core 2 should observe the store to `x` (since it observed the store to `y` which happened after the store to `x`). However, on an nMCA system, simply ordering `i2` before `i4` and `i5` before `i7` is not enough to ensure that core 2 sees the store to `x` (i.e., to forbid the outcome `r1=1,r2=1,r3=0`). This is because under nMCA, the store `i1` to `x` can become visible to core 1 without becoming visible to core 2 as well. To forbid the outcome `r1=1,r2=1,r3=0`, the fences need to be cumulative. If the fence `i3` were cumulative, it would order `i1` with respect to `i4` in addition to ordering `i2` with respect to `i4`. Thus, if core 2 observes `i4`, it must now observe `i1` as well, which will result in in `i7` returning 1. The cumulative ordering of `i4` combined with the orderings enforced by `i6` (`i6` need not be cumulative for this particular test) is enough to forbid the outcome `r1=1,r2=1,r3=0` on an nMCA architecture. A lack of cumulative fences was one of the major issues with the draft specification of the RISC-V MCM (see TriCheck [TML+17] for details). There are multiple flavours of cumulativity; these are explained in prior work such as Sarkar et al. [SSA+11].

A.7 Virtual Memory

Modern processors implement virtual memory to isolate processes running at the same time from each other, as well as to abstract the size of physical memory away from the programmer. The virtual to physical address mappings in a system can have ramifications for MCM correctness. Consider the litmus test in Figure A.10, taken from COATCheck [LSMB16]. If `x` and `y` map to different physical addresses, then

Core 0	**Core 1**
(i1) [x] ← 1	(i3) [y] ← 2
(i2) r1 ← [y]	(i4) r2 ← [x]
Outcome: r1=2, r2=1	

Figure A.10: Code for litmus test illustrating how virtual to physical address mappings can affect MCM behaviour, taken from COATCheck [LSMB16]. If `x` and `y` map to different physical addresses, then the specified outcome is allowed under SC. However, if `x` and `y` map to the same physical address, then the specified outcome is forbidden under SC.

the outcome `r1=2,r2=1` is allowed under SC. If `x` and `y` map to the same physical address, though, then the test essentially reduces to a single-address litmus test. In that case, the outcome `r1=2,r2=1` is forbidden under SC.

The vast majority of MCM work ignores virtual memory considerations, but a number of papers have examined the issue. Romanescu et al. [RLS10] were the first to differentiate between MCMs for virtual addresses and those for physical addresses. They also developed approaches for the efficient dynamic verification (Section 2.3.5) of virtual address memory consistency. COATCheck [LSMB16] built on PipeCheck [LPM14] (Section 2.4) to enable both the address translation-aware formal specification of microarchitectural orderings and the formal verification of such ordering specifications for novel virtual memory-aware "enhanced" litmus tests. It also coined the term "transistency model" to denote the superset of MCMs that captures all translation-aware sets of ordering rules. TransForm [HTM20] creates a vocabulary for formally specifying transistency models and develops a candidate transistency model for x86. TransForm can also automatically synthesize enhanced litmus tests.

A.8 Summary

Microarchitectural features such as those in this section all affect the execution of memory operations in ways which break SC. Outlawing all of these features would result in an unacceptable performance hit, so architects strongly prefer to keep these features in their hardware designs. There are two ways to do so. One is to implement the reordering speculatively (Section 2.1.2) i.e., roll back the execution of the reordered memory instructions when the reordering would become programmer-visible, and then re-execute them. The other is to have the processor MCM be a weak MCM (Section 2.1.3) that allows the reorderings to be programmer-visible. Depending on the types of reorderings employed by the designers, the specification of the processor's weak MCM (Section 2.3.1) may need to include certain classes of attributes. For instance, the use of nMCA write atomicity in a processor requires defining a notion of cumulativity for the fences in its ISA-level MCM.

Most processors today (including all commercial ones) have chosen to implement weak MCMs [Int13, AMD13, SPA94, ARM13, IBM13, RIS19] rather than adopt a purely speculative approach. However, the two approaches are not mutually exclusive. Architects may choose to use an MCM for their ISA that allows some reorderings and forbids others, and then speculatively implement some of the forbidden reorderings. For instance, Intel and AMD architectures have a consistency model of TSO (which forbids Load-Load reordering), but speculatively reorder pairs of load instructions [Int13, AMD13].

Bibliography

[AAA$^+$15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for TSO and PSO. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 353–367, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[AAJ$^+$19] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. Optimal stateless model checking for reads-from equivalence under sequential consistency. *Proc. ACM Program. Lang.*, 3(Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)), October 2019.

[AAJL16] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 134–156, Cham, 2016. Springer International Publishing.

[ABB64] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks. Architecture of the IBM System/360. *IBM J. Res. Dev.*, 8(2):87–101, April 1964.

[ABD$^+$15] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 577–591. ACM, 2015.

[ACDJ01] Mark Aagaard, Byron Cook, Nancy A. Day, and Robert B. Jones. A framework for microprocessor correctness statements. In *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Livingston, Scotland, UK, September 4-7, 2001, Proceedings*, pages 433–448, 2001.

[AG96] Sarita Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[AH90] Sarita Adve and Mark Hill. Weak ordering: a new definition. *17th International Symposium on Computer Architecture (ISCA)*, 1990.

[AJM$^+$00] Mark Aagaard, Robert B. Jones, Thomas F. Melham, John W. O'Leary, and Carl-Johan H. Seger. A methodology for large-scale hardware verification. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, pages 263–282, 2000.

[AKNT13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 512–532, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 141–157, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Alg17] Ben Algaze. Software is increasingly complex. That can be dangerous., 2017. https://www.extremetech.com/computing/259977-software-increasingly-complex-thats-dangerous.

[Alp85] F. B Alpern, B; Schneider. Defining liveness. *Information processing letters*, 1985.

[AMD12] AMD. Revision guide for amd family 10h processors, 2012. `https://www.amd.com/system/files/TechDocs/41322_10h_Rev_Gd.pdf`.

[AMD13] AMD. AMD64 architecture programmer's manual. 2013.

[AMM$^+$18] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 405–418. Association for Computing Machinery, 2018.

[AMSS10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. *CAV*, 2010.

[AMSS11] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–44, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[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, July 2014.

[ARM11] ARM. Cortex-A9 MPCore, programmer advice notice, read-after-read hazards. ARM Reference 761319., 2011. `http://infocenter.arm.com/help/topic/com.arm.doc.uan0004a/UAN0004A_a9_read_read.pdf`.

[ARM13] ARM. *ARM Architecture Reference Manual*, 2013.

[ASA18] Johnathan Alsop, Matthew D. Sinclair, and Sarita V. Adve. Spandex: A flexible interface for efficient heterogeneous coherence. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 261–274. IEEE Press, 2018.

[BA08] Hans-J. Boehm and Sarita Adve. Foundations of the C++ concurrency memory model. *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[BBDL98] Ilan Beer, Shoham Ben-David, and Avner Landver. On-the-fly model checking of RCTL formulas. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, pages 184–194, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[BCM$^+$92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. *Inf. Comput.*, 98(2):142–170, June 1992.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Computer Aided Verification*, pages 68–80, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[BDW16] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *43rd Annual Symposium on Principles of Programming Languages (POPL)*, 2016.

[BEH$^+$17] Christopher J. Banks, Marco Elver, Ruth Hoffmann, Susmit Sarkar, Paul Jackson, and Vijay Nagarajan. Verification of a lazy cache coherence protocol against a weak memory model. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, FMCAD '17, page 60–67, Austin, Texas, 2017. FMCAD Inc.

[BGMW17] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 155–168. Association for Computing Machinery, 2017.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BK18] Armin Biere and Daniel Kröning. SAT-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 277–303. Springer International Publishing, Cham, 2018.

[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 ACM SIGPLAN 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$^+$11a] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, , and Tjark Weber. Mathematizing C++ concurrency. *38th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2011.

[Bos11b] Pradip Bose. Power wall. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1593–1608. Springer US, Boston, MA, 2011.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

[BT17] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. In *38th Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[BT18] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer International Publishing, Cham, 2018.

[Büc90] J. Richard Büchi. On a decision method in restricted second order arithmetic. In Saunders Mac Lane and Dirk Siefkes, editors, *The Collected Works of J. Richard Büchi*, pages 425–435. Springer New York, 1990.

[BVR$^+$12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012.

[Cad15a] Cadence Design Systems, Inc. *JasperGold Apps Command Reference Manual*, 2015.

[Cad15b] Cadence Design Systems, Inc. *JasperGold Apps User's Guide*, 2015.

[Cad16] Cadence Design Systems, Inc. *JasperGold Engine Selection Guide*, 2016.

[CDH$^+$15] Eduard Cerny, Surrendra Dudani, John Havlicek, Dmitry Korchemny, et al. *SVA: The Power of Assertions in SystemVerilog*. Springer, 2015.

[CDS$^+$14] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 269–284. Association for Computing Machinery, 2014.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, page 52–71, Berlin, Heidelberg, 1981. Springer-Verlag.

[CFH$^+$03] Edmund Clarke, Ansgar Fehnker, Zhi Han, Bruce Krogh, Olaf Stursberg, and Michael Theobald. Verification of hybrid systems based on counterexample-guided abstraction refinement. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 192–207, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[CGJ$^+$00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *12th International Conference on Computer Aided Verification (CAV)*, 2000.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[CHV18] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. Introduction to model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 1–26. Springer International Publishing, Cham, 2018.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[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. *PACT*, 2011.

[CMP08] K. Chen, S. Malik, and P. Patra. Runtime validation of memory ordering using constraint graph checking. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 415–426, 2008.

[Col92] William W. Collier. *Reasoning About Parallel Architectures.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[com16] comododragon. Stores are not working, 2016. https://github.com/ucb-bar/vscale/issues/13.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158. Association for Computing Machinery, 1971.

[Coq04] The Coq development team. *The Coq proof assistant reference manual, version 8.0.* LogiCal Project, 2004. http://coq.inria.fr.

[CSG02] Prosenjit Chatterjee, Hemanthkumar Sivaraj, and Ganesh Gopalakrishnan. Shared memory consistency protocol verification against weak memory models: Refinement via model-checking. In *14th International Conference on Computer Aided Verification (CAV)*, 2002.

[CSW18] Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, power, arm, and c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 211–225. Association for Computing Machinery, 2018.

[CTMT07] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk enforcement of sequential consistency. *34th International Symposium on Computer Architecture (ISCA)*, 2007.

[CV17] Soham Chakraborty and Viktor Vafeiadis. Formalizing the concurrency semantics of an LLVM fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 100–110. IEEE Press, 2017.

[CVS$^+$17] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP), 2017.

[DBG18] Mike Dodds, Mark Batty, and Alexey Gotsman. Compositional verification of compiler optimisations on relaxed memory. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 1027–1055, Cham, 2018. Springer International Publishing.

[DGY$^+$74] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[Dij02] Edsger W. Dijkstra. Cooperating sequential processes. In *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, page 65–138. Springer-Verlag, Berlin, Heidelberg, 2002.

[diy12] The diy development team. *A don't (diy) tutorial, version 5.01*, 2012. http://diy.inria.fr/doc/index.html.

[DLCO09] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 85–96. Association for Computing Machinery, 2009.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[DSB86] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. *13th International Symposium on Computer Architecture (ISCA)*, 1986.

[EF18] Cindy Eisner and Dana Fisman. Functional specification of hardware via temporal logic. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 795–829. Springer International Publishing, Cham, 2018.

[EL87] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, 8(3):275 – 306, 1987.

[Elv15] Marco Elver. *TSO-CC Specification*, 2015. http://homepages.inf.ed.ac.uk/s0787712/res/research/tsocc/tso-cc_spec.pdf.

[F$^+$20] Facebook AI Research et al. Pytorch, 2020. `https://pytorch.org/`.

[FGP+16] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 608–621, 2016.

[Fos15] Harry D. Foster. Trends in functional verification: A 2014 industry study. *52nd Design Automation Conference (DAC)*, 2015.

[FSP+17] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Paris)*, pages 429–442, January 2017.

[GFV99] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? *26th International Symposium on Computer Architecture (ISCA)*, 1999.

[GGH91] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *In Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, 1991.

[GJS+14] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[GKM+15] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pages 635–646, 2015.

[Gle98] Andrew Glew. MLP yes! ILP no! *ASPLOS Wild and Crazy Ideas*, 1998.

[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.

[GNBD16] R. Guanciale, H. Nemati, C. Baumann, and M. Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 38–55, May 2016.

[God97] Patrice Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In Orna Grumberg, editor, *Computer Aided Verification*, pages 476–479, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Goo89] J. R. Goodman. Cache consistency and sequential consistency. Technical report, SCI Committee, March 1989. Tech Report 61.

[Goo20] Google. Tensorflow, 2020. `https://www.tensorflow.org/`.

[GSV$^+$10] N. Goulding, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, J. Babb, M. B. Taylor, and S. Swanson. GreenDroid: A mobile application processor for a future of dark silicon. In *2010 IEEE Hot Chips 22 Symposium (HCS)*, pages 1–39, 2010.

[Gup17] Aarti Gupta. Lecture slides on Satisfiability Modulo Theories DPLL(T), 2017.

[Hac14] Mark Hachman. Intel finds specialized TSX enterprise bug on Haswell, Broadwell CPUs, 2014. http://www.pcworld.com/article/2464880/intel-finds-specialized-tsx-enterprise-bug-on-haswell-broadwell-cpus.html.

[Hil98] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *Computer*, 31(8):28–34, August 1998.

[HLR10] Tim Harris, James Larus, and Ravi Rajwar. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, page 289–300. Association for Computing Machinery, 1993.

[HP89] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, page 477–498. Springer-Verlag, Berlin, Heidelberg, 1989.

[HP17] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[HR19] Mark D. Hill and Vijay Janapa Reddi. Accelerator-level parallelism. *CoRR*, abs/1907.02064, 2019.

[HTM20] Naorin Hossain, Caroline Trippel, and Margaret Martonosi. TransForm: Formally specifying transistency models and synthesizing enhanced litmus tests. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 874–887, 2020.

[HVML04] Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, and Juin-Yeu Joseph Lu. TSOtool: A program for verifying memory systems using the memory consistency model. In *31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[HWS+16] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[IBM13] IBM. *Power ISA Version 2.07*, 2013.

[IEE10] IEEE. IEEE standard for Property Specification Language (PSL). *IEEE Std 1850-2010 (Revision of IEEE Std1850-2005)*, April 2010.

[IEE13] *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*, 2013.

[Int13] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2013. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[Int20] Intel. 6th generation Intel processor family specification update, November 2020.

[Isa20] The Isabelle development team. *The Isabelle/Isar Reference Manual*, 2020. https://isabelle.in.tum.de/dist/Isabelle2020/doc/isar-ref.pdf.

[ISO11a] ISO/IEC. Programming Languages – C. International standard 9899:2011, ISO/IEC, 2011.

[ISO11b] ISO/IEC. Programming Languages – C++. International standard 14882:2011, ISO/IEC, 2011.

[Jac12] Daniel Jackson. Alloy analyzer website, 2012. http://alloy.mit.edu.

[KCZ92] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *19th Annual International Symposium on Computer Architecture*, 1992.

[KHF+19] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[KLSV17] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[KRV19] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 96–110. Association for Computing Machinery, 2019.

[KV99] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, pages 172–183, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[KV20] Michalis Kokologiannakis and Viktor Vafeiadis. HMC: Model checking for hardware memory models. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1157–1171. Association for Computing Machinery, 2020.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computing*, 28(9):690–691, 1979.

[Lea20] The Lean development team. *The Lean reference manual*, 2020. https://leanprover.github.io/reference/index.html.

[Lee19] Timothy B. Lee. Autopilot was active when a Tesla crashed into a truck, killing driver, 2019. https://arstechnica.com/cars/2019/05/feds-autopilot-was-active-during-deadly-march-tesla-crash/.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[LOM15] Yunsup Lee, Albert Ou, and Albert Magyar. Z-scale: Tiny 32-bit RISC-V systems, 2015. https://riscv.org/wp-content/uploads/2015/06/riscv-zscale-workshop-june2015.pdf.

[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.

[LSG$^+$18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association.

[LSG19] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. A formal analysis of the NVIDIA PTX memory consistency model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 257–270. Association for Computing Machinery, 2019.

[LSMB16] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. COATCheck: Verifying Memory Ordering at the Hardware-OS Interface. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[Lus15] Daniel Lustig. *Specifying, Verifying, and Translating Between Memory Consistency Models*. PhD thesis, Princeton University, Princeton, NJ, USA, 2015.

[LVK$^+$17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[LWPG17] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. Automated synthesis of comprehensive memory model litmus test suites. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 661–675. ACM, 2017.

[Mag16] Albert Magyar. Verilog version of z-scale, 2016. https://github.com/ucb-bar/vscale.

[Man07] Jeremy Manson. Java concurrency: Roach motels and the Java memory model, 2007. `http://jeremymanson.blogspot.com/2007/05/roach-motels-and-java-memory-model.html`.

[Mar18] Margaret Martonosi. New metrics and models for a post-ISA era: Managing complexity and scaling performance in heterogeneous parallelism and Internet-of-Things. *SIGMETRICS Perform. Eval. Rev.*, 46(1):20–20, June 2018.

[MGJ$^+$19] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 370–384. Association for Computing Machinery, 2019.

[MHAM10] Sela Mador-Haim, Rajeev Alur, and Milo M K. Martin. Generating litmus tests for contrasting memory consistency models. In *22nd International Conference on Computer Aided Verification (CAV)*, 2010.

[MHAM11] Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. Litmus tests for comparing memory consistency models: How long do they need to be? In *Proceedings of the 48th Design Automation Conference*, DAC '11, page 504–509. Association for Computing Machinery, 2011.

[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. In *24th International Conference on Computer Aided Verification (CAV)*, 2012.

[MLM20] Yatin A. Manerkar, Daniel Lustig, and Margaret Martonosi. RealityCheck: Bringing modularity, hierarchy, and abstraction to automated microarchitectural memory consistency verification. *CoRR*, abs/2003.04892, 2020.

[MLMG18] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Aarti Gupta. PipeProof: Automated memory consistency proofs for microarchitectural specifications. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 788–801. IEEE Press, 2018.

[MLMP17] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLCheck: Verifying the memory consistency of RTL designs. In *50th International Symposium on Microarchitecture (MICRO)*, 2017.

[MLPM15] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. CCICheck: Using μhb graphs to verify the coherence-consistency interface. In *48th International Symposium on Microarchitecture (MICRO)*, 2015.

[Moo06a] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[Moo06b] G. E. Moore. Progress in digital integrated electronics [technical literature, copyright 1975 IEEE. reprinted, with permission. technical digest. international electron devices meeting, IEEE, 1975, pp. 11-13.]. *IEEE Solid-State Circuits Society Newsletter*, 11(3):36–37, 2006.

[MS09] A. Meixner and D.J. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2009.

[MSC+01] Milo M. K. Martin, Daniel J. Sorin, Harold W. Cain, Mark D. Hill, and Mikko H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *34th International Symposium on Microarchitecture (MICRO)*, 2001.

[MTL$^+$16] Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. *CoRR*, abs/1611.01507, 2016.

[ND13] Brian Norris and Brian Demsky. CDSChecker: Checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2013.

[Nik19] Rishiyur S. Nikhil. RISC-V ISA formal specification, 2019. `https://github.com/riscv/ISA_Formal_Spec_Public_Review/blob/master/Forvis.md`.

[NRZ$^+$15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Commun. ACM*, 58(4):66–73, March 2015.

[NSHW20] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence, Second Edition.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[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 *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 73–86. Association for Computing Machinery, 2015.

[OD17] Peizhao Ou and Brian Demsky. Checking concurrent data structures under the C/C++11 memory model. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, page 45–59. Association for Computing Machinery, 2017.

[ONS18] Nicolai Oswald, Vijay Nagarajan, and Daniel J. Sorin. Protogen: Automatically generating directory cache coherence protocols from atomic specifications. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 247–260. IEEE Press, 2018.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2009.

[PFD$^+$18] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. In *45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2018.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[PLBN05] Michael Pellauer, Mieszko Lis, Don Baltus, and Rishiyur S. Nikhil. Synthesis of synchronous assertions with guarded atomic actions. In *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005), 11-14 July 2005, Verona, Italy, Proceedings*, pages 15–24, 2005.

[PLSS17] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: Decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[PMP$^+$16] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 614–630. Association for Computing Machinery, 2016.

[PNAD95] Fong Pong, Andreas Nowatzyk, Gunes Aybay, and Michel Dubois. Verifying distributed directory-based cache coherence protocols: S3.mp, a case study. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *EURO-PAR '95 Parallel Processing*, pages 287–300, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, page 46–57, USA, 1977. IEEE Computer Society.

[PP18] Nir Piterman and Amir Pnueli. Temporal logic and fair discrete systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 27–73. Springer International Publishing, Cham, 2018.

[RCD$^+$16] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with ISA-Formal. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 42–58, Cham, 2016. Springer International Publishing.

[RIS15] RISC-V Foundation. RISC-V in Verilog, 2015. https://riscv.org/2015/09/risc-v-in-verilog/.

[RIS19] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20190608-Base-Ratified*, June 2019.

[RLS10] Bogdan F. Romanescu, Alvin R. Lebeck, and Daniel J. Sorin. Specifying and dynamically verifying address translation-aware memory consistency. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[RPA97] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. *SPAA*, 1997.

[Rup20] Karl Rupp. 48 years of microprocessor trend data, 2020. https://github.com/karlrupp/microprocessor-trend-data/blob/master/48yrs/48-years-processor-trend.pdf.

[SD16] Tyler Sorensen and Alastair F. Donaldson. Exposing errors related to weak memory in GPU applications. In *37th Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2016.

[SGNR14] Daryl Stewart, David Gilday, Daniel Nevill, and Thomas Roberts. Processor memory system verification using DOGReL: a language for specifying end-to-end properties. *DIFTS*, 2014.

[Shi08] Anand Lal Shimpi. AMD's B3 stepping Phenom previewed, TLB hardware fix tested, 2008. https://www.anandtech.com/show/2477.

[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.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 2nd edition, 2006.

[SPA92] SPARC International. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[SPA94] 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)*, 1988.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER microprocessors. *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[Str19] Tom Strickx. How Verizon and a BGP optimizer knocked large parts of the Internet offline today, 2019. https://blog.cloudflare.com/how-verizon-and-a-bgp-optimizer-knocked-large-parts-of-the-internet-offline-today/.

[SVRM15] Pramod Subramanyan, Yakir Vizel, Sayak Ray, and Sharad Malik. Template-based synthesis of instruction-level abstractions for SoC verification. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, pages 160–167, Austin, TX, 2015. FMCAD Inc.

[TLM18a] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate: Automated synthesis of hardware exploits and security litmus tests. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 947–960, 2018.

[TLM18b] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *CoRR*, abs/1802.03802, 2018.

[TML$^+$17] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. TriCheck: Memory model verification at the trisection of software, hardware, and ISA. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[TML$^+$18] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Full-stack memory model verification with TriCheck. *IEEE Micro*, 38(03):58–68, May 2018.

[TS08] Babu Turumella and Mukesh Sharma. Assertion-based verification of a 32 thread SPARC CMT microprocessor. In *Proceedings of the 45th Annual Design Automation Conference (DAC)*, 2008.

[VCAD15] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular deductive verification of multiprocessor hardware designs. In *27th International Conference on Computer Aided Verification (CAV)*, 2015.

[Wal18] Steven Walton. AMD Ryzen Threadripper 2990wx & 2950x review, 2018. https://www.techspot.com/review/1678-amd-ryzen-threadripper-2990wx-2950x/page2.html.

[WBBD15] John Wickerson, Mark Batty, Bradford M. Beckmann, and Alastair F. Donaldson. Remote-scope promotion: Clarified, rectified, and verified. *SIGPLAN Not.*, 50(10):731–747, October 2015.

[WBSC17] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.

[Wol81] Pierre Wolper. Temporal logic can be more expressive. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, SFCS '81, page 340–348, USA, 1981. IEEE Computer Society.

[WS19] Emma Wang and Yakun Sophia Shao. Die photo analysis, 2019. http://vlsiarch.eecs.harvard.edu/research/accelerators/die-photo-analysis/.

[YL05] Ping Yeung and K. Larsen. Practical assertion-based formal verification for SoC designs. In *2005 International Symposium on System-on-Chip*, pages 58–61, Nov 2005.

[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.

[ZTM$^+$18] Hongce Zhang, Caroline Trippel, Yatin A. Manerkar, Aarti Gupta, Margaret Martonosi, and Sharad Malik. ILA-MCM: Integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2018.