# Modular Verification of Hardware Systems

by

Muralidaran Vijayaraghavan

B.Tech., Indian Institute of Technology, Madras (2006)
S.M., Massachusetts Institute of Technology (2009)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

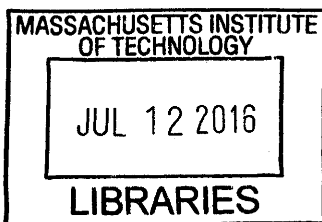
© Massachusetts Institute of Technology 2016. All rights reserved.

Author ....... Signature redacted ...............
Department of Electrical Engineering and Computer Science
February 17, 2016

Certified by.... Signature redacted ..................
Arvind
Professor
Thesis Supervisor

Certified by.. Signature redacted ....................
Adam Chlipala
Associate Professor
Thesis Supervisor

Accepted by .. Signature redacted ..................
Professor Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 12 2016
LIBRARIES
ARCHIVES

## Modular Verification of Hardware Systems

by

Muralidaran Vijayaraghavan

Submitted to the Department of Electrical Engineering and Computer Science
on February 17, 2016, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

## Abstract

As hardware systems are becoming bigger and more complex, it is becoming increasingly harder to design and reason about these systems in a monolithic fashion. While hardware is often designed in a modular manner, its verification is rarely performed modularly. Moreover, any modular refinement to an existing system requires a full system verification to guarantee correctness, even if only a few components of the system have been refined.

In this thesis, we present a new framework for modular verification of hardware designs written in the Bluespec language. That is, we formalize the idea of components in a hardware design, with well-defined input and output channels; and we show how to specify and verify components individually. For verifying a full system, we show how the proofs of its components can be composed, treating the components as black-boxes and not looking beyond their specifications.

As a demonstration of this methodology, we verify a fairly realistic implementation of a multicore shared-memory system with two types of components: memory system and processor, with machine-checked proofs in the Coq proof assistant. Both components include nontrivial optimizations, with the memory system employing an arbitrary hierarchy of cache nodes that communicate with each other concurrently, and with the processor doing speculative execution of many concurrent read operations. Nonetheless, we prove that the combined system implements sequential consistency. To our knowledge, our memory-system proof is the first machine verification of a cache-coherence protocol parameterized over an arbitrary cache hierarchy, and our full-system proof is the first machine verification of sequential consistency for a multicore hardware design that includes caches and speculative processors.

We also embed the Bluespec language in the Coq proof assistant and formalize its modular semantics, enabling a verification engineer to obtain machine-checked proofs for Bluespec designs using our framework.

Thesis Supervisor: Arvind
Title: Professor

Thesis Supervisor: Adam Chlipala
Title: Associate Professor

# Acknowledgments

As my long and adventurous journey as a PhD student comes to an end, there are several people to whom I would like to express my gratitude and appreciation. These few paragraphs of acknowledgments would not do justice to all the help and support they have given me over all these years – for that, I apologize in earnest in advance.

First and foremost, I would like to thank Professor Arvind, my advisor and mentor, for guiding me throughout grad school. He not only allowed me, but also actively encouraged me to explore different topics for research. His tutelage was instrumental in shifting my research interests from building systems to something more theoretical. He introduced me to the world of functional languages and rigorous semantics which culminated in my thesis being about formal verification of hardware systems. He taught me an important lesson about research: communicating the ideas is as important as solving the problem. He also showed me that just rigor and completeness are not sufficient for a good practical theory, but also simplicity.

Second, I want to thank my co-advisor, Associate Professor Adam Chlipala, for all the technical guidance he has provided to my thesis. I was bumbling around with a paper-and-pencil proof of correctness of a cache coherence protocol when I first met him, and he encouraged me to formalize my ideas in Coq. He also convinced and guided me to develop the cache coherence protocol proof into a full fledged thesis on formal hardware verification.

I also want to thank Professor Joel Emer, who was my supervisor when I interned at Intel, Hudson, and Dr. Ekanadham Kattamuri, who supervised me at IBM Research, TJ Watson. Both of them gave me valuable insights that helped me nurture my ideas. I still remember the post-lunch walks I took with Dr. Ekanadham during my internship at IBM Research where we debated on the merits of formal verification using theorem provers for hardware systems.

Staying in grad school for a long time has its advantages: I had the pleasure of working with several people closely on various projects, even in topics unrelated to my thesis. I have been collaborating with Joonwon Choi for the last couple of years

to develop a hardware verification framework, *Kami*, based on my thesis. Another colleague, Ben Sherman has also been involved in this project for a while. The library we use in Kami borrows heavily from the code written by Dr. Ben Delaware. Dr. Nirav Dave has been my friend and mentor throughout my stay in grad school; he helped me with my initial formulation of the cache coherence problem and also introduced me to various formal verification techniques. Dr. Alfred Man Cheuk Ng was another friend and mentor during the early part of my grad school; it was due to his constant skepticism about the correctness of a cache coherence protocol used in the graduate Computer Architecture class at MIT (6.823) that I became interested in this problem. When I interned at Intel, I worked closely with Dr. Michael Pellauer and Nikhil Patil, both of whom had tremendous influence in the way I approach a problem. The two other colleagues I had the most interactions with are Richard Euhler and Sizhuo Zhang. Both of them are extremely rigorous and systematic; they have always been available to discuss my half-baked ideas and point out the issues in them. I also started working with Andy Wright and Thomas Bourgeat towards the end of my tenure as a grad student, and they have given me valuable feedback on my dissertation. I would also like to thank Dr. Thomas Braibant for his advice on using Coq and for the discussions on embedding a hardware description language in Coq.

Life in grad school would not have been as rewarding for me without all my friends and social circle in and around MIT. They are now like a family away from home to me. I would like to thank my office mates Shuotao, Sangwoo, Ming and Sungjin and my former office mates Myron and Abhinav for the constant bantering that kept me entertained throughout. Shuotao was also the "man with the car"; he never once refused to give me a ride or help me move heavy furniture in his SUV. I am forever grateful to Sungjin for his help during my stay in Korea. I would like to specially thank my friends outside CSAIL, Nikhil Galagali, Rajan Udwani and Nishanth Mundru, for all the help they had provided around the time of my thesis defense. Without their help there is no way I could have completed my move out of MIT's Ashdown house and caught the flight back home the day after my defense! Rajan has also accompanied me in numerous ski and snowboarding trips, and has

been a regular attendee of one of my favorite pastimes, the board game nights in our dorms. I would also like to thank Lee Drahushuk for organizing the board game nights, and Sai Gautam and Siddharth who were also regular attendees. I am also thankful to Dr. Karthik Venkataraman who was my former roommate, and has been my friend through thick and thin. Finally, I would like to thank my extended social circle in the greater Boston area and my undergrad friends for tagging with me in numerous trips.

Life at MIT would have been a lot harder without all the help I received from others. Gita, Prof. Arvind's wife, has always been very kind to me and really helped me to get over difficult times. Nikhil Galagali's mother had come to Cambridge during the last few months leading up to my thesis defense and had been feeding me with delicious home-cooked food for all those months! My uncle Naren, who lives in western Massachusetts, has been treating me as family and my uncle Siva had flown all the way from California to help me settle down right after I landed in Boston.

I would not have been interested in research if not for my grandpa, Mr. Annamalai, and my middle school physics teacher Prof. Ananthan. They were the ones who made me fall in love with math at that age, which influenced me to pursue grad studies in computer science right after undergrad. I would like to thank my wife, Radhika, for all the support she has given me from the time we had known each other. And a big thanks to my parents, Dr. Vijayaraghavan and Dr. Nagalatha, who have always stood by me all through these years and encouraged me to pursue my dreams. I would like to specially thank my mom for all the times she visited me in Cambridge, taking care of daily chores and cooking delicious food for me and my friends whenever she visited. Finally, I want to thank my grandma, Mrs. Sarada, for everything she has done for me. I dedicate this thesis to my grandparents.

# Contents


**1 Introduction** **15**

**I Hardware System Implementations and Specifications** **20**

**2 Background: Hardware Designs using the Bluespec Language** **25**

2.1 Examples of Specifications in the Bluespec Language . . . . . . . . . 25

2.1.1 Specification of a FIFO buffer . . . . . . . . . . . . . . . . . . . 26

2.1.2 Specification of a Stream Counter . . . . . . . . . . . . . . . . . 28

2.1.3 Instances of Modules . . . . . . . . . . . . . . . . . . . . . . . 30

2.1.4 Composition of Modules . . . . . . . . . . . . . . . . . . . . . 31

2.2 Semantics of Bluespec . . . . . . . . . . . . . . . . . . . . . . . . . 31

2.3 Synthesizing Bluespec Designs . . . . . . . . . . . . . . . . . . . . . 33

2.4 Advantages of Bluespec for Designing Hardware . . . . . . . . . . . . 35

2.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

**3 Multiprocessor System Specification and Implementation** **37**

3.1 Sequential Consistency . . . . . . . . . . . . . . . . . . . . . . . . . 37

3.1.1 Abstracting the ISA . . . . . . . . . . . . . . . . . . . . . . . . 38

3.1.2 Specifying Sequential Consistency . . . . . . . . . . . . . . . . . 39

3.2 Respecifying Sequential Consistency with Communication . . . . . . 41

3.2.1 The Decoupled Processor . . . . . . . . . . . . . . . . . . . . . 42

3.2.2 Buffers between Processors and Memory . . . . . . . . . . . . 42

3.2.3 Atomic Memory: Instantaneous Memory + Decoupling Buffers 46

3.2.4 Decoupled Multiprocessor System . . . . . . . . . . . . . . . . 46
3.3 Speculative Out-of-Order Processor . . . . . . . . . . . . . . . . . . . . . 47
3.3.1 Components of the Speculative Out-of-Order Processor . . . . 47
3.4 Full-system Verification of the Multiprocessor . . . . . . . . . . . . . 56
3.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

**4 Implementing the Memory Subsystem Using Coherent Caches 59**
4.1 Structure of Hierarchy of Caches . . . . . . . . . . . . . . . . . . . . . 60
4.2 Cache Nodes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.2.1 Abstractions Used in the Cache Nodes . . . . . . . . . . . . . 62
4.2.2 Naming Conventions used in the Cache Nodes . . . . . . . . . 63
4.2.3 L1 cache . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.2.4 Memory . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
4.2.5 Internal Cache . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.2.6 Subtle Design Decisions in the Caches . . . . . . . . . . . . . 78
4.3 Network Between the Caches . . . . . . . . . . . . . . . . . . . . . . 80
4.4 Design for Provability . . . . . . . . . . . . . . . . . . . . . . . . 82
4.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

**II Verifying Hardware Systems using Labeled Transition Systems 86**

**5 Background: Verification Techniques for Hardware Systems 89**
5.1 Hardware Verification using Model-Checking . . . . . . . . . . . . . . 89
5.2 Hardware Verification using Theorem Provers . . . . . . . . . . . . . 92
5.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

**6 Modular Semantics of Bluespec 95**
6.1 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
6.2 Meaning of Module Composition . . . . . . . . . . . . . . . . . . . . 99
6.3 Modular Semantics . . . . . . . . . . . . . . . . . . . . . . . . . 102

6.3.1 Semantics for Expressions . . . . . . . . . . . . . . . . . . . . 102
6.3.2 Semantics for Actions . . . . . . . . . . . . . . . . . . . . . . . . 103
6.3.3 Semantics for Modules . . . . . . . . . . . . . . . . . . . . . . 105
6.4 Trace-Refines Relation . . . . . . . . . . . . . . . . . . . . . . . . . 113
6.5 Properties of Trace-Refines Relation . . . . . . . . . . . . . . . . . . 115
6.5.1 Decomposing a Multistep . . . . . . . . . . . . . . . . . . . . . 119
6.6 Weak Implements Relation . . . . . . . . . . . . . . . . . . . . . . . 123
6.7 Limitations of the Semantics . . . . . . . . . . . . . . . . . . . . . . 124
6.8 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126

**7 Verifying the Complete Multiprocessor System Implementation 127**
7.1 Removing Speculative Loads in the Labels of Atomic Memory . . . . 130
7.2 Refinement Relation between Speculative Out-of-Order Processor and Instantaneous Processor . . . . . . . . . . . . . . . . . . . . . . . . . 133
7.2.1 Speculative Out-of-Order Processor Implements Decoupled Processor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
7.2.2 Decoupled Processor Implements Instantaneous Processor . . . 135
7.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138

**8 Verifying the Memory Subsystem 139**
8.1 Soundness of a Hierarchy of Coherent Caches w.r.t. Atomic Memory . 139
8.1.1 Cache Invariants . . . . . . . . . . . . . . . . . . . . . . . . . . 147
8.1.2 Notation used in the Invariants and the Proofs . . . . . . . . . 147
8.1.3 Formal Specification of Cache Invariants . . . . . . . . . . . . 149
8.2 Deadlock-freedom in a Hierarchy of Coherent Caches . . . . . . . . . 162
8.3 Livelock-freedom in a Hierarchy of Coherent Caches . . . . . . . . . . 167
8.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 181

**9 Conclusions and Future Work 183**

**A Functions Used in Defining Semantics 187**
A.1 Extracting Values from Finite Register Maps . . . . . . . . . . . . . . 187

# List of Figures


2-1 Module *FIFO*: A two-element FIFO . . . . . . . . . . . . . . . . . . . . 26

2-2 Module *StreamCounter'*: Another example of a Bluespec module . . . 29

3-1 Module $P_{\text{inst}}$: An instantaneous processor . . . . . . . . . . . . . . . . . 39

3-2 Module $M_{\text{inst}}$: An instantaneous memory . . . . . . . . . . . . . . . . 40

3-3 Components of $SC_{\text{dec}}$, *i.e.,* sequential consistency with communication 41

3-4 $P_{\text{dec}}$: A decoupled processor serving as specification of $P_{\text{so}}$ . . . . . . 43

3-5 Module $M'_{\text{wrap}}$: Rules for processing load and store requests from the buffers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

3-6 Module $P'_{\text{so}}$: The speculative out-of-order processor module (the reorder buffer and the branch predictor are separate modules) . . . . . 51

4-1 Example instantiation of the memory subsystem . . . . . . . . . . . . 60

4-2 Register Arrays in each cache node . . . . . . . . . . . . . . . . . . . 62

4-3 L1 cache . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

4-4 Module L1': Module containing the transition rules of the L1 cache . 67

4-5 The augmented state changes for request entries in *cRqs* . . . . . . . 68

4-6 Memory . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71

4-7 Module $M'$: The module containing the transition rules of the memory 71

4-8 The augmented state changes for request entries in *cRqs* in the memory 73

4-9 Internal Cache . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76

4-10 Module $\text{L}_{\text{int}}$': Module containing the transition rules of the $\text{L}_{\text{int}}$ cache 77

6-1 Primitive Module given by $[\![P^1_{\text{inst}} + M_{\text{inst}}]\!]$ . . . . . . . . . . . . . . . 101

6-2 Modules $m_1$ and $m_2$ showing the limitations of our modular semantics 125

7-1 Overall structure of the proof of the full system . . . . . . . . . . . . 128

8-1 Stages for a request $r$ from a processor to an L1 cache $c$ . . . . . . . . 169

8-2 Stages for a request $r$ from a cache to its parent $c$ (which is not memory) 170

8-3 Partial order for stages representing evicting a cache line . . . . . . . 171

8-4 Partial order for stages representing upgrading the cache state and downgrading the other children, for a $\langle \mathsf{WaitSt}, l \rangle$-request $r = \langle c', a, y, M \rangle$ in the *cRqs* buffer of an internal cache . . . . . . . . . . . . . . . . . . 172

8-5 Stages for a request $r$ from a cache to memory $m$ . . . . . . . . . . . 173

8-6 Partial order for stages representing upgrading the cache state and downgrading the other children, for a $\mathsf{WaitSt}$-request $r = \langle c', a, y, M \rangle$ in the *cRqs* buffer of memory . . . . . . . . . . . . . . . . . . . . . . . 174

8-7 Stages for a request from a parent to its child $c$ . . . . . . . . . . . . 174

8-8 Partial order for stages representing downgrading a cache line for processing request $r = \langle \mathsf{Req}, a, I \rangle$ from the parent. . . . . . . . . . . . . . 175

# Chapter 1

# Introduction

Hardware systems are inherently concurrent and highly non-deterministic. Modern processors, for example, have several highly concurrent cores communicating with a cache-coherent, distributed-memory system. The goal of this work is to provide a framework for full verification of such complex hardware systems.

Modularity has long been understood as a key property for effective design and verification of such complex systems [72]. For the designer, it allows a separation of concerns, increasing robustness by allowing the behavior encapsulated by a modular boundary to be realized by multiple implementations, any of which may be dropped into the system safely. It also allows a greater parallelization of human design effort, improving development time. Similarly, verification is simplified, as modular interface agreements provide a natural lemma structure. They lead us towards a decomposition of the whole-system verification task into lemmas about subsystems, which can be composed in a black-box manner to produce full-system theorems.

The traditional notion of modularity in hardware is closely tied to synchronous Finite-State Machines (FSMs). The word "synchronous" implies the presence of a notion of clock cycles, and during each clock cycle, a module reads its current state, reads its inputs, produces its outputs and updates its states. Multiple modules are composed by connecting inputs of one module to outputs of another module while ensuring that a causal relation is obeyed for reading the inputs and producing outputs (*i.e.,* there is no "combinational cycle" or a cycle with inputs and outputs with no

state elements.)

The biggest problem with using FSM abstraction is that it requires global reasoning. For example, if we change the timing of an adder module so that it takes 2 clock cycles instead of 1, the whole system is likely to break. Modular reasoning and verification is close to impossible unless all the details of the rest of the system are known. A state transition in a full-system FSM is complex, taking into account the transitions in every module that comprises the FSM. This significantly restricts the kinds of refinements that can be performed on a module.

Bluespec [6,12] presents an alternative to using FSM-based modularity for designing hardware systems. In this methodology, a hardware system is described in terms of modules, each of which contains *guarded atomic actions* or *rules*. These rules are atomic state transitions reading the state of the module and updating it. The behavior of a module obeys *one-rule-at-a-time* semantics, where any state reached by the system can be explained by a sequence of rule executions, one at a time [5,44,45]. The commercial Bluespec compiler synthesizes such descriptions into circuits which are competitive in performance to those generated from synchronous FSM descriptions (like Verilog, VHDL, *etc.*) [4,26,30–33,44,52,67,68,75–77].The compiler automatically generates a scheduler circuit to schedule several "non-conflicting" rules, *i.e.,* rules with disjoint read and write sets for registers, to execute concurrently. Such a scheduling will not violate the one-rule-at-a-time semantics. Thus, reasoning about the system becomes easier as one has to simply reason about atomic transitions. The burden of concurrent execution of atomic actions is left to the compiler and the correctness of the compiler can be proven once for all programs. Because of these advantages, we chose to use Bluespec-like specifications for the design of our modules.

A system in Bluespec is composed of several modules. These modules communicate with each other via *methods* that employ asynchronous handshaking protocols. The direct support for asynchronous handshaking protocols also enables a much wider variety of modular refinement in Bluespec compared to designing using synchronous FSMs.

Each rule in Bluespec, by design, is very similar to an inference rule in operational

semantics. While the semantics of a single-module Bluespec program looks just like a simple transition system which is well understood, it gets tricky to provide the semantics of multiple modules. One option is to flatten the module hierarchy, but this approach is antithetical to modular reasoning. In order to formalize the semantics of a composition of modules in Bluespec, we provide modular semantics for Bluespec modules, which are independent of the context in which the module is used. These semantics are isomorphic to those of *Labeled Transition Systems (LTSes)* [65], a well-studied approach used to express the operational semantics of many process calculi. LTSes have successfully been used in concurrent software modeling. We use the labels of an LTS to model the communication in Bluespec between modules via methods calls, and the state transitions in Bluespec are transitions in LTSes.

As our unified notion of specification and proof, we adopt *trace refinement*, which captures when one concurrent system can produce only those observable behaviors that another system could also produce. Each of our realistic hardware components is associated with a simpler "reference implementation" that serves as a specification, and we prove that each realistic component refines its spec. Thereafter, it is sound to use the simpler spec component in reasoning about the behavior of the system.

As a challenging case study involving realistic hardware designs, we focus on *verifying that a particular infinite family of multicore, shared-memory systems implements sequential consistency.* The proof is parametrized over an unknown number of processors connected to an arbitrary memory hierarchy with an unknown number of caches in an unknown number of layers (e.g., L1, L2).

Each key component of our case-study system has a simple specification. The cache hierarchy allows multiple caches to cache a location simultaneously, doing cache lookups, and communicating with other caches to keep the locations "coherent." The specification for such a complex system is a monolithic memory that responds instantly and atomically to each load or store request. Similarly, the spec for the processor executes instructions in order, atomically with no speculation. But the actual processor speculatively executes all non-store instructions, sending multiple load requests to the memory concurrently (without waiting for load responses before

executing other instructions). The meat of the verification is in showing that each optimized component refines its simpler counterpart. After each such proof, we may in effect substitute the optimized version for the simple version in a black-box way within a design, with a full guarantee of soundness. Thus, we are able to verify the components of a modern processor against general formal interfaces, enabling mixing and matching of different realizations of each interface, without doing any new proofs that peek beneath abstraction boundaries.

All our proofs about soundness are mechanized in the Coq proof assistant, and they are modular in the sense of allowing further optimization to either processor or memory without needing to touch the proof of the other. We also verify the deadlock-freedom property mechanically in the Coq proof assistant for the cache hierarchy and give a paper-and-pencil proof for the livelock-freedom property.

In terms of developing the general verification framework, we embed the Bluespec language in Coq and formally specify its modular semantics inside Coq using the PHOAS approach [22]. We also formally show the isomorphism between the modular semantics of Bluespec and those of LTSes. While ideally this must have preceded our case study involving verification of the multicore system, we had performed the case study first as a proof of concept before we embedded the Bluespec language in Coq. Future Bluespec designs can be formally verified using this framework.

The main contributions of this thesis are:

- A general *modular verification methodology* applicable to hardware algorithms, based on *labeled transition systems.*

- A *deep embedding of the Bluespec language in Coq*, a formal specification of its *modular semantics and its relation to labeled transition systems.*

- A complete detailed design of an invalidation-based cache-coherence protocol for distributed memories comprised of arbitrary cache hierarchies.

- The *first machine-checked proof* of soundness of the cache hierarchy design with respect to an atomic memory specification and a machine-checked proof of deadlock-freedom.

- The *first machine-checked proof* of an implementation of sequential consistency involving speculative processors and cache-coherent distributed memory.

**Thesis Organization:** We organize the thesis into two parts. Part I is concerned with giving a background overview of designing hardware modules using Bluespec. We then give the Bluespec designs for various multiprocessor systems, starting with a simple *Sequential Consistency* specification all the way to a complex system consisting of speculative out-of-order processors connected to a hierarchy of coherent caches. In Part II, we start with a background overview of various hardware verification techniques. We contrast the approach against the more widely used approach in hardware verification involving model-checking. We then specify the modular semantics of Bluespec and formally verify that the complex multiprocessor system is actually an implementation of the simpler system. We end this part with a discussion of the work to be done in terms of improving the verification framework and building a library of formally verified hardware components using this technique.

# Part I

# Hardware System Implementations and Specifications

# Organization

How are systems verified? On one hand, one can specify properties that the system should obey. For example, one can verify whether a program ever executes a divide-by-zero instruction. However, just verifying a given set of properties does not guarantee that the system is behaving correctly. One has to specify a complete set of properties that a system should obey and then verify if the system obeys all these properties in order to have a guarantee that the system is correct. For example, for a sorting function, it is not enough to specify that the resulting list/array is sorted; one has to also specify that the resulting list/array is a permutation of the input list/array.

Another important criterion that is desirable for verifying large systems is that of composability. Verifying a system composed of several components can scale only if we can treat the individual components as black boxes and use the properties of the individual components to get the property for the full system. But ad-hoc listing of properties of the individual components may not be strong enough to guarantee that they can be composed into a full-system property.

One way to ascertain that all the properties that a system should obey have been stated is to specify a simpler system and prove that the "behavior" of the more complex system is contained within the simpler system. If the behavior of the system $A$ is contained in the behavior of another system $B$, then the $A$ is said to *implement* $B$, or that $A$ is a refinement of $B$. Hence, system $A$ can be called as the *implementation* for the *specification*, *i.e.,* system $B$. This approach naturally lends itself to composable verification because if the behavior of an implementation is contained within the behavior of its specification, then the specification can be replaced by its implementation in any context!

Intuitively, the behavior of the system is the externally observable interactions that the system has with its environment. If the behavior of one system is contained within another, then the external environment cannot tell the former system from the latter. This thesis is focused on verification of hardware systems. In order to verify if a complex hardware system is correct, we first specify the desired behavior in terms of a simpler hardware system and then verify if the complex system implements the simpler system. We will be formalizing the notion of "containment of behaviors," "implements" or "refinement," *etc.*, in the second part of this dissertation.

We will use the concrete problem of multiprocessor verification as our running example. Multiprocessors are one of the most complex yet common hardware designs, making them an ideal case study for full-system hardware verification. They can be broadly divided into two components: the processor subsystem and the memory subsystem. Each of these subsystems are complex systems in their own right. In Chapter 3, we will give the specification of a multiprocessor system in terms of the simple sequential consistency specification as described by Lamport [56]. We will then refine the processors and the memory subsystems progressively with Chapter 3 dealing exclusively with refinements of the processor subsystem and Chapter 4 dealing with refinements of the memory subsystem.

As our first refinement of sequential consistency, we will discuss a simple decoupled multiprocessor system, where unlike Lamport's sequential consistency specification, the processors and the memory are separated by buffers. We will then refine the processor further by designing a speculative out-of-order processor which executes instructions, sends memory requests and receives their responses out of order.

We will refine the memory subsystem by designing a hierarchy of coherent caches in Chapter 4. These caches employ a directory-based coherence protocol. We will give the complete detailed specification of the protocol in this chapter.

We will be using a stylized version of the Bluespec SystemVerilog (BSV) Hardware Description Language (HDL) [12] for describing the modules. While Bluespec provides many syntactic conveniences over more conventional HDLs like Verilog and VHDL, and has a powerful type system unlike the other HDLs, the primary reason

why we chose Bluespec is because designs written in Bluespec naturally lend themselves to verification, as we shall see in the next chapter (Chapter 2).

# Chapter 2

# Background: Hardware Designs using the Bluespec Language

The goal of this chapter is to give an informal overview of the Bluespec language which is the language we use for specifying all the hardware designs in this dissertation. This chapter is organized as follows. In Section 2.1, we give examples of modules and their compositions in the Bluespec language. In Section 2.2, we discuss their semantics informally, and in Section 2.3, we describe the synthesis procedure for generating hardware circuits from the Bluespec programs, again informally. Finally, in Section 2.4, we will discuss why designing hardware systems using Bluespec makes the verification of systems inherently easier.

## 2.1 Examples of Specifications in the Bluespec Language

In this section we will give examples of hardware module specifications in the Bluespec language to give a quick overview of the syntax of programs in Bluespec and an intuition about their semantics. While we will be omitting the specification of types throughout the dissertation, types need to be specified in real Bluespec programs.

```
Module:
Regs d[2]({_}), v[2]({False});

Meth enq(a):
when(¬v[0]) ⇒
    d[0] := a;
    v[0] := True;

Meth pop:
when(v[0] ∨ v[1]) ⇒
    v[1] := v[0];
    d[1] := d[0];
    v[0] := False;
    return(if (v[1]) then d[1] else d[0]);
```

Figure 2-1: Module *FIFO*: A two-element FIFO

### 2.1.1 Specification of a FIFO buffer

Figure 2-1 shows the specification of a simple module implementing a two-element FIFO buffer.

The statement **Regs** $d[2](\{_\}), v[2](\{\mathsf{False}\})$ specifies that the *FIFO* module has two register arrays $d$ and $v$. Both are two-element arrays. In general, an array $x$ is specified using $x[\mathsf{NUM}]$ where $\mathsf{NUM}$ is the number of elements in the array. The values in parentheses specify the initial values of the respective registers. The special value of _ stands for undefined value, indicating that the corresponding register is not initialized. Register arrays must be initialized with a list of values. If the initial value is inside curly braces $\{\ldots\}$, then it indicates that all elements of the array are initialized with that value. Register arrays can also be initialized with a constant list of values. For example, **Regs** $x[2](x_0)$ initializes the $i^{\text{th}}$ register with the value given by the $i^{\text{th}}$ element of the list $x_0$. Multi-dimensional arrays are written as $x[\mathsf{NUM}_1]\ldots[\mathsf{NUM}_n](\{\ldots\{x_0\}\ldots\})$ which initializes all the elements of array $x$ to value $x_0$.

The two register arrays in the *FIFO* module, *viz.,* $d$ for data and $v$ for valid, represent the data present in the buffer and the validity of the corresponding data, respectively. The register $v[i]$ indicates that the corresponding data in register $d[i]$ has been enqueued earlier and yet to be popped or moved from slot $i$.

The specification of registers is followed by the specification of *methods*. Methods

are the means by which modules communicate with each other. A module *calls* methods defined by another module, and in this call, it can pass zero or more arguments. Our *FIFO* module defines two methods, *enq* and *pop*, which can be called by other modules. A called method returns a value to the caller, and in addition, may or may not change the state of the module defining that method. The return value is specified using the statement **return**($x$) which returns a value $x$ to the caller. If the method does not contain any return statement, it does not return any value to the caller.

The first method $enq(a)$ specifies the action of enqueuing a value $a$ into the buffer into slot 0. The **when** keyword specifies a guard over an action. A guarded action can be performed only when the guard is true. The caller of the method has to ensure that the guard of the method's action is true before calling the method. In the *enq* example, the guard prevents enqueue from taking place if the buffer already has an element in slot 0. This is ascertained by checking the validity of the data in slot 0 by reading the register $v[0]$.

Stripping away the guard in the body of *enq* method leaves us with the actual action corresponding to the method. An action is composed of multiple sub-actions separated by semicolons. In our *enq* example, the first action $d[0] := a$ updates the data register $d[0]$ with the enqueued value, and the second action $v[0] := \mathsf{True}$ sets the valid register $v[0]$ indicating that the data in slot 0 is valid. In general, a statement $r := e$ represents an update of register $r$ with expression $e$. The enqueue method does not return any value.

The second method *pop* removes the oldest element from the buffer and returns it. The action in *pop* method is guarded by a condition that the buffer has at least one element. The condition that the buffer is not empty is asserted if and only if either $v[0]$ or $v[1]$ is not set as is explained below. The *pop* method does not have any parameters.

The body of *pop* method performs four sub-actions, the first three being register updates and the last sub-action being a return action. It first shifts the data and valid values from the enqueue-slot (slot 0) to slot 1, so that future pops can simply

read slot 1. Then, it unsets $v[0]$ indicating that the data in slot 0 is no longer valid, thereby allowing later enqueues to happen in slot 0. It finally returns either the value originally present (*i.e.,* before it got updated because of the action in *pop* method's body) in slot 1 if it is valid, or the value originally present in slot 0 (which has to be valid if slot 1 is not valid, since we already checked that one of $v[0]$ or $v[1]$ is valid in the guard of the *pop* method).

The methods in the *FIFO* module have showed the different kinds of expressions in the Bluespec language. An expression can either be a register read, a method argument access, a constant, or a complex expression formed by combining one or more expressions using arithmetic or logical operators (for example $(v[0] \vee v[1])$ in the guard of the *pop* method or the return value of *pop* method, **if** $(v[1])$ **then** $d[1]$ **else** $d[0]$).

A general module is a collection of *registers* with their initial values, *rules* and *methods*. The FIFO buffer module, which was discussed in this section, does not have any rules, but the next section discusses a stream counter module which has a rule.

### 2.1.2 Specification of a Stream Counter

We will now design a stream counter module which interacts with two modules $f_{\text{in}}$, an input buffer; and $f_{\text{out}}$, an output buffer. The stream counter pops a value from the input buffer and enqueues that value into the output buffer. Simultaneously, the stream counter also increments one of the two counter registers $c_0$ or $c_1$ present in the module.

The data being popped from the input buffer is a pair $\langle v, b \rangle$ where $v$ specifies the message and $b$ is a boolean value indicating which counter to increment. If the boolean is True, counter $c_0$ is incremented; otherwise, counter $c_1$ is incremented. Figure 2-2 shows the specification for the stream counter module.

This module introduces the concept of a **Rule**. A rule comprises of an action that can execute at any time. If the action is guarded, then the guard must be true for the rule to execute.

In the stream counter example, the action constituting the **Rule**$\langle$*Process*$\rangle$ is com-

**Module:**
**Regs** $c_0(0), c_1(0)$;

**Rule** *Process*:
$\textbf{let}\langle v, b\rangle = f_{\text{in}}.pop$;
**if** $(b)$ **then** $c_0 := c_0 + 1$ **else** $c_1 := c_1 + 1$;
$f_{\text{out}}.enq(\langle v, b\rangle)$;

**Meth** *getCount*$_0$():
$\textbf{return}(c_0)$;

**Meth** *getCount*$_1$():
$\textbf{return}(c_1)$;

Figure 2-2: Module *StreamCounter′*: Another example of a Bluespec module

posed of two sub-actions. The first sub-action is a call to method $f_{\text{in}}.pop$, without passing any arguments. This call pops the oldest element from the input buffer. This method is guarded by an assertion that the buffer $f_{\text{in}}$ is not empty. If the assertion is false, then the method cannot be called inside **Rule**⟨*Process*⟩, and hence **Rule**⟨*Process*⟩ cannot fire. The return value of this call is a tuple whose elements are bound to the variables $v$ and $b$ via the **let** expression. Any expression can be bound to temporary variables. In hardware terms, these variables correspond to wires (as opposed to registers) holding the assigned values.

We make extensive use of pattern matching in **let** expressions. In particular, if we are binding expressions which are tuples to a matching tuple of variables on the left, then each element of the tuple expression is bound to a variable that is present in the same position in the left pattern as the element of the tuple expression. We can also bind nested tuples. For example, $\textbf{let}\langle x, \langle y, z\rangle\rangle = \langle 2, \langle 3, 4\rangle\rangle$ will bind $x$ to 2, $y$ to 3 and $z$ to 4.

When a constant is present in the left pattern of a **let** expression, the overall action of which the **let** expression is a part is guarded with a constraint that the expression returns the specified constant. For example, the statement $\textbf{let}\langle v, \mathsf{True}\rangle = f_{\text{in}}.pop; \ldots$ is equivalent to $\textbf{let}\langle v, b\rangle = f_{\text{in}}.pop; \textbf{when}(b) \ldots$.

The second action in **Rule**⟨*Process*⟩ is a conditional action. If the value of the variable $b$ is True, then counter $c_0$ is incremented. Otherwise, counter $c_1$ is incremented. We may omit the **else** clause in the (**if** (...) **then** ... **else** ...) action if

there is no action corresponding to the **else** clause.

The last action in the rule is a call to method $f_{\text{out}}.enq$ with the argument $\langle v, b\rangle$, which enqueues the data into the output buffer. This method is guarded by the condition that $f_{\text{out}}$ buffer is not full, and **Rule**$\langle$*Process*$\rangle$ will not fire unless that is true.

### 2.1.3 Instances of Modules

The *FIFO* module defined in Figure 2.1.1 is a reusable library component, in the sense that this module can be used in any design which requires a two-element FIFO buffer. In fact, the $f_{\text{in}}$ and $f_{\text{out}}$ modules connected to the stream counter module can themselves be "instances" of this *FIFO* module. Technically, there is a difference between a module (like *FIFO*) and an *instance* of the module (like $f_{\text{in}}$ or $f_{\text{out}}$). But, the same effect can be achieved by creating a renamed copy of the module, *i.e.,* a copy of the module where the registers and methods are renamed. For example, for instance $f_{\text{in}}$, we can prepend "$f_{\text{in}}$." to all register and method names, resulting in $f_{\text{in}}.enq$ and $f_{\text{in}}.pop$ methods, and $f_{\text{in}}.d$ and $f_{\text{in}}.v$ register arrays. For the purposes of this dissertation, all modules are unique, *i.e.,* there is no separate concept of instantiations of modules; each module is an instance of itself. Moreover, the names used in the modules' registers and the methods are all globally distinct. Prepending the "instance" names to the registers, rules and methods of each module will create globally distinct names as long as the instance names are globally unique, which will be assumed throughout. For the sake of convenience, we omit the prefix for registers, rules and methods whenever the module being discussed is clear from the context.

Since this dissertation uses the example of a multiprocessor system, it is useful to instantiate several "copies" of the same module, such as the processor. We usually use $M^i$ to denote the $i^{\text{th}}$ copy of module $M$. The registers, rules and methods defined inside $M$ will also get the superscript $^i$ in that case.

### 2.1.4 Composition of Modules

Modules are composed using the + operator. For instance, we can create a new module which is a composition of the stream buffer module with the input and output buffers as shown below:

$$\mathit{StreamCounter} \triangleq \mathit{StreamCounter}' + f_{\text{in}} + f_{\text{out}}$$

The meaning of the + operator applied on two modules is straightforward – it effectively creates a new module which has all the registers, rules and methods of the two individual modules. But the bodies of all methods which are called as well as defined in the newly created module are inlined at their call sites, and are removed or *hidden* in the new module. For instance, in the above composition, $f_{\text{in}}$ and $f_{\text{out}}$ both have methods *enq* and *pop*. However, module *StreamCounter'* calls $f_{\text{in}}.\mathit{pop}$ and $f_{\text{out}}.\mathit{enq}$. Thus, module *StreamCounter*, formed by composing *StreamCounter'* with $f_{\text{in}}$ and $f_{\text{out}}$, effectively exposes only methods $f_{\text{in}}.\mathit{enq}$ and $f_{\text{out}}.\mathit{pop}$.

While the meaning of the + operator can be understood in terms of merging two modules, we do not actually create a new module whenever the + operator is encountered. In other words, the + operator is not a syntactic sugar, but rather a fundamental operator in our syntax. In Chapter 6, we will be giving modular semantics for individual modules and semantics for composing two modules using the + operator. We will also be showing that the semantics for composing two modules matches the semantics of the merged module described above.

## 2.2 Semantics of Bluespec

We will now informally describe the behavior of modules specified in Bluespec.

Let us start with a simpler system consisting of just one module, and no methods defined by this module. Informally, the behavior of such a module is the set of all possible sequences of state transitions that the module undergoes, starting from the initial state. Each state transition is caused by the execution of a single rule in the module, which reads the module's registers and updates them. It appears as if exactly

one rule *fires* in each transition step, and this behavior is aptly named *one-rule-at-a-time* semantics. Rules in Bluespec are called *atomic actions* for the same reason. In a given state, if there are multiple rules that can fire, then any one rule can be chosen to fire non-deterministically. This gives rise to multiple transition sequences defining the behavior of a module.

All the register reads in the action corresponding to a rule happen before any register update happens, even if a register read appears syntactically after a write to the same register. It will appear as if all the sub-actions within a rule's action happen concurrently during a transition, with all the register reads happening at the beginning of the transition and all the register updates happening at the end. Updating the same register multiple times in an action makes that action illegal.

An important concept related to Bluespec actions is that of a guard. Any action (or sub-action) can be guarded. If an action consists of several guarded sub-actions, then the guards of all these sub-actions must is true for the rule to be able to fire. This creates a complication in the presence of conditional actions. Only those guards of conditional actions which are in the "taken-branch" of the conditional need to be true.

Notice that the only way a module can communicate with the external environment is via methods that are called by the methods or rules of the module or via the methods defined by the module. We have restricted the module to not have any defined methods, so the only externally observable behavior of a module is via the called methods. In fact, the behavior of a module is not defined by the transition sequence (as the module's state is not visible externally) but by the sequence of methods called by the module along with their arguments and return values.

If the module defines methods, then its behavior must include the calling of these methods by the external environment. Thus, the behavior includes the sequence of methods called by the module as well as the sequence of methods that can be called by the external environment (with arguments and return values for both called and defined methods). We will define this notion formally in Chapter 6 in Part II.

The easiest way to understand the semantics of a composition of modules is con-

sidering a module formed by flattening the module definitions. This global module contains all the registers present in the individual modules, and the definitions of all the called methods are inlined at the places of their calls, binding the return values of the methods to the variables that binds the method calls.

While it is easy to understand semantics by flattening the module definitions, it is important to give semantics of a module independent of the context in which the module resides. This enables us to develop reusable modular proofs for various components. This is the main topic of Chapter 6 in Part II.

Once the methods are inlined, the composition of modules contains a set of rules, each rule specifying an action (which comprises of several sub-actions). An action is illegal if there are two sub-actions within that action, both of which write to the same register or call the same method.

## 2.3 Synthesizing Bluespec Designs

Synthesizing Bluespec programs into hardware circuits is a fundamental requirement for using the Bluespec language to describe hardware. While we will informally describe how Bluespec programs are synthesized into hardware circuits in this section, the synthesis problem is orthogonal to the verification problem which is the focus of this dissertation.

Each rule in the system generates a combinational circuit producing values corresponding to the register updates. The **let**-bindings create circuits for the bound expressions, which can be reused within the rule. Whenever a rule fires, then the corresponding registers are updated using this combinational circuit.

While the semantics of Bluespec dictates that each rule in the overall system fires one-at-a-time, actually synthesizing a hardware circuit where only the circuit corresponding to one rule is active during any clock-cycle is very inefficient, performance-wise. If there are two "independent" or *non-conflicting* rules, *i.e.,* rules that do not access the same set of registers, then these two rules can fire concurrently without breaking the one-rule-at-a-time semantics. The resulting state transition will be as if

the two individual transitions happened one after the other. The actual commercial Bluespec compiler takes advantage of this observation to fire multiple rules, almost always more than just two rules, concurrently in the same hardware clock cycle.

In order to fire multiple rules concurrently in one clock cycle, the concept of *conflict* must be well defined. One simple definition of conflict is as follows: two rules are conflicting only if they both *access*, *i.e.,* read or write, the same register, and at least one of the accesses is a write access. If two rules do not access any common register, or only read a common register, then these rules are non-conflicting. If two non-conflicting rules fire concurrently in one clock-cycle, then their state updates will be as if the rules fired one after the other. In this definition of conflict, either order of firing the rules will lead to the same state updates. This definition of conflict is similar to that in database transactions and transactional memory [9, 42, 87].

Instead of defining conflict between a pair of rules, one can have a more sophisticated criterion for deciding if a set of two or more rules can all fire together concurrently in one clock cycle. A set of rules can fire concurrently if there exists a permutation of this set such that for any pair of rules $r_1$ and $r_2$ in the set, whenever $r_1$ occurs before $r_2$ in the permutation, the registers that $r_1$ writes are disjoint from the registers that $r_2$ reads. If all the rules in such a set fire concurrently in one clock cycle, with all the register reads happening at the beginning of the clock cycle, and all the register updates happening at the end of the clock cycle, then the final state updates will be as if the rules happened one after the other according to the permutation. This notion of conflict has been studied in great detail by Hoe *et al.* [3, 44, 45].

The commercial Bluespec compiler creates a *scheduler* to schedule the firing of each rule based on the more sophisticated notion of conflict described above. In terms of the synthesized hardware, every clock cycle, the scheduler decides which rules can fire during that cycle. Conceptually, the scheduler behaves as follows: Each rule in the overall system is statically given a (unique) priority which can be assigned either automatically or supplied by the designer. During each cycle, the scheduler enables those rules whose guards are true to fire, starting with the highest priority rule. A rule whose guard is true is enabled to fire if and only if it has no conflict with

another higher priority rule that has also been enabled to fire. If the designer does not supply any priority, the Bluespec compiler tries to assign priorities automatically to maximize the number of rules being fired every clock cycle. Since these priorities are assigned statically, the scheduler converts a nondeterministic Bluespec specification into a deterministic hardware circuit.

Another important question regarding synthesis is the treatment of methods. In the discussion of semantics of Bluespec programs in Section 2.2, we inlined the methods at the points of their calls. But synthesis has to be modular for various reasons. One reason is to reduce the complexity of the scheduler circuit. Another, more fundamental reason is that methods are the only mechanism by which a program written in Bluespec (either a single module or a composition of several modules) communicates with an external environment, and these external interactions cannot be inlined. Thus, one has to specify how methods are synthesized into hardware circuits.

Each method gets synthesized into several input and output wires for the hardware circuit synthesized for a Bluespec module. Each method has an input *enable* signal which is a boolean signal indicating whether the method is called during the current clock cycle. A method also has input *argument* wires that carry the values being passed to the method in the current clock cycle. It has an output *guard* signal which is also a boolean signal returning the value of the guard of the action that constitutes the body of the method. Finally, a method has output *return* wires that carry the value returned by the method in the current clock cycle.

Another complication for modular synthesis in the presence of methods is that two modules can call each other's methods. A comprehensive discussion of the issues and the solution for modular synthesis in the presence of such "mutual" calls is presented by Vijayaraghavan *et al.* [86].

## 2.4 Advantages of Bluespec for Designing Hardware

We chose to use Bluespec specifications for designing hardware as opposed to traditional netlist-based specifications (like Verilog or VHDL). The main advantage of

Bluespec programs is that they lend themselves to verification. As we will be seeing in Part II, transition systems are usually verified using simulation techniques which relate a single transition in one system to a sequence of transitions in another system. In Bluespec programs, state transitions are not tied to hardware clock cycles. Instead, state transitions are expressed as rules, and multiple rules are scheduled together concurrently to constitute one hardware clock cycle as discussed in Section 2.3. So, instead of using simulation-based verification techniques on a complex rule composed of several rules firing concurrently, one only has to analyze a single rule, significantly simplifying the verification problem.

## 2.5 Conclusion

In this chapter we gave an introduction to designing hardware using Bluespec. We specified the semantics of Bluespec programs and discussed how Bluespec programs can be synthesized into hardware circuits, informally. In the next chapter, we will design a complex multiprocessor system in several stages, starting with a simple system implementing sequential consistency as defined by Lamport [56].

# Chapter 3

# Multiprocessor System Specification and Implementation

Modern multiprocessor systems are extremely complex. They are made up of several major components each employing optimizations like caching, speculation, *etc.* Though the actual system's implementation is very complex, its behavior can usually be understood in terms of a simpler specification.

In this chapter we will start with a simple specification of multiprocessor systems, *viz.,* the sequential consistency specification. Over the course of this chapter, we will design increasingly complex implementations of this specification.

## 3.1 Sequential Consistency

We will now specify the simplest notion of correctness of a multiprocessor system, *viz.,* the sequential consistency specification. Before we embark on that task, we will first abstract the instruction set architecture (ISA) of the processors. Parameterizing the processor over the ISA enables design of the "core logic" of the processor independent of the ISA. Further, it enables instantiation of complex and realistic ISAs which have long-latency operations that showcase performance gains of the out-of-order processor.

### 3.1.1 Abstracting the ISA

Each instruction executed by a processor performs a combination of the following operations: (a) update the program counter, (b) update architectural registers, (c) perform memory operations, and/or (d) halt execution. Thus, the ISA can be abstracted by using two functions: $dec(s, pc, i)$, which returns the "decoded" form of an instruction $i$, specifying the combination of the above operations that the instruction performs, and a function $exec(s, pc, d)$ which returns the updated state and the next program counter. We specify both these functions without giving their implementations below.

$dec(s, pc, i)$ : The legal decoded instruction forms, *i.e.,* the outputs of *dec*, are (a) $\langle \mathsf{Nm}, x\rangle$, for an operation not accessing memory, $x$ specifying the actual operation being performed, along with the registers being read and written, constants, *etc.*; (b) $\langle \mathsf{Ld}, va, x\rangle$, for a memory load from word address $va$, and $x$ specifying the register being updated; $\langle \mathsf{St}, va, v\rangle$, for a memory store of value $v$ to word address $va$; and $\langle Halt\rangle$, for a "halt" instruction.

$exec(s, pc, d)$ : The legal arguments to *exec* are the current registers and the program counter, and an encoding of both a decoded instruction and any relevant responses from the memory system. The third argument of *exec* is $\langle \mathsf{Nm}, x\rangle$, $\langle \mathsf{St}\rangle$ or $\langle \mathsf{Ld}, x, v\rangle$. The tuples $\langle \mathsf{Nm}, x\rangle$ and $\langle \mathsf{St}\rangle$ are obtained from the decoded instruction, and the value $x$ in $\langle \mathsf{Ld}, x, v\rangle$ is obtained from the decoded instruction while the value $v$ is the data present in the requested memory location.

The advantage of abstracting the ISA using functions is that the same functions can be used in the specification as well as in all its refinements. Therefore, we are not verifying if a particular operation (like add or subtract) is implemented correctly; instead we are instantiating the same operations in the specification and its refinements thus ensuring that each implementation has the same behavior as the specification with respect to these operations.

**Module**:
**Regs** $pc(pc_0), s(s_0)$;

**Rule** *NonMemory*:

**let** $i = getInst(pc)$;
**let** $\langle \mathsf{Nm}, x\rangle = dec(s, pc, i)$;
**let** $\langle s', pc'\rangle = exec(s, pc, \langle \mathsf{Nm}, x\rangle)$;
$s := s'$;
$pc := pc'$;

**Rule** *Halt*:

**let** $i = getInst(pc)$;
**let** $\langle \mathsf{Halt}\rangle = dec(s, pc, i)$;
*halt*;

**Rule** *Load*:

**let** $i = getInst(pc)$;
**let** $\langle \mathsf{Ld}, va, r_v\rangle = dec(s, pc, i)$;
**let** $v = M_{\text{inst}}.ldRq(va)$;
**let** $\langle s', pc'\rangle = exec(s, pc, \langle \mathsf{Ld}, r_v, v\rangle)$;
$s := s'$;
$pc := pc'$;

**Rule** *Store*:

**let** $i = getInst(pc)$;
**let** $\langle \mathsf{St}, va, v\rangle = dec(s, pc, i)$;
$M_{\text{inst}}.stRq(va, v)$;
**let** $\langle s', pc'\rangle = exec(s, pc, \langle \mathsf{St}\rangle)$;
$s := s'$;
$pc := pc'$;

Figure 3-1: Module $P_{\text{inst}}$: An instantaneous processor

### 3.1.2 Specifying Sequential Consistency

We are now in a position to specify sequential consistency using the abstract ISA. The sequential consistency specification that we give in this section is exactly as is described by Lamport [56]. According to Lamport, a multiprocessor system is sequentially consistent if the behavior it exhibits while running a multi-threaded program can be reasoned about by performing the following operation repeatedly: Pick any one processor and atomically execute the next instruction in the thread residing in that processor, including performing load and store operations, while all other processors remain idle.

In order to define the sequential consistency specification, we first need to specify an *instantaneous processor* $P_{\text{inst}}$, which executes all instructions, including memory loads and stores, instantaneously. Figure 3-1 shows the design of such a processor module.

```
Module:
Regs mem[NUM](mem_0);

Meth ldRq^1(va):
return(mem[va]);
...
Meth ldRq^n(va):
return(mem[va]);

Meth stRq^1(va, v):
mem[va] := v;
...
Meth stRq^n(va, v):
mem[va] := v;
```

Figure 3-2: Module $M_{\text{inst}}$: An instantaneous memory

The state of the instantaneous processor consists of the program counter *pc* initialized to $pc_0$, and the architectural registers $s$, initialized to $s_0$. In every rule of the instantaneous processor, an instruction is fetched using *getInst* based on the current program counter, decoded and executed using *dec* and *exec* functions, respectively, and finally the program counter is updated to point to the next instruction, and one or more architectural registers are updated.

In case of a halt instruction, an external method *halt* is called and the architectural registers and the program counter remain the same. In the case of memory instructions (loads or stores), the processor calls external methods. The instantaneous processor is connected to an *instantaneous memory* $M_{\text{inst}}$ which looks up data for load requests or updates the data for store requests instantaneously. The return value of the load is used to update the architectural registers and the program counter.

Figure 3-2 shows the design of the instantaneous memory module containing a register array *mem*[NUM] and having two kinds of methods: *ldRq* for reading a memory value and *stRq* for storing into the memory. There are several copies of the same pair of methods (each copy of the pair is known as a *port* in describing a memory), each connected to a different processor.

Using the specifications of the instantaneous processor and the instantaneous memory, we can define the specification of sequential consistency as a composition of several atomic processors and an instantaneous memory as shown below. Note that $P^i_{\text{inst}}$ represents the $i^{\text{th}}$ instantaneous processor, which calls $ldRq^i$ or $stRq^i$ rather than

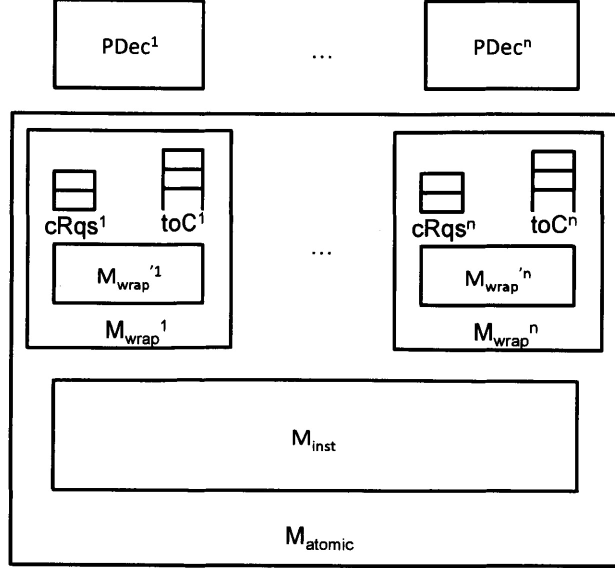



Figure 3-3: Components of $SC_{\text{dec}}$, *i.e.,* sequential consistency with communication

just *ldRq* or *stRq*.

**Definition 1.** $SC \triangleq P^1_{\text{inst}} + \ldots + P^n_{\text{inst}} + M_{\text{inst}}$

## 3.2 Respecifying Sequential Consistency with Communication

The first refinement that we introduce over the sequential consistency specification in Definition 1 is introducing buffers between the processors and the memory. Here the responses from the memory to the processor are decoupled from the requests from the processor to the memory. We first show the specification of this decoupled processor, followed by the specification of the buffers separating the memory and the processor, and its composition with the instantaneous memory. Figure 3-3 shows the components of a specification of sequential consistency with communication, using the decoupled processors $P_{\text{dec}}$, the buffers separating the memory and the processor (*cRqs* and *toC*), and the instantaneous memory $M_{\text{inst}}$. Each of the components will be described next.

### 3.2.1 The Decoupled Processor

Figure 3-4 shows such a decoupled processor. A state of $P_{\text{dec}}$ is given by three registers: (a) $s$ which gives the state of the architectural registers, (b) $pc$ which gives the program counter, and (c) $w$ which is a Boolean flag indicating whether the processor is blocked waiting for a response from the memory system. The registers are initialized to $s_0, pc_0$ and False, respectively.

On a non-memory or halt instruction, the processor behaves exactly as in the instantaneous processor. These two instructions can be executed only when the processor is not in wait state (*i.e.*, the $w$ register is False).

On a load instruction, the processor executes **Rule**⟨*LoadRq*⟩ followed by **Rule**⟨*LoadRs*⟩. In **Rule**⟨*LoadRq*⟩, the processor sends a load request to the memory. The processor then goes into a wait state waiting for a load response back from the memory. As discussed before, no rule other than **Rule**⟨*LoadRs*⟩ (or **Rule**⟨*StoreRs*⟩) can execute when the processor is waiting. Moreover **Rule**⟨*StoreRs*⟩ can execute only when a store response is obtained from the memory, which will not be the case if a load request has been sent to the memory. Once it receives the load response from the memory, it executes **Rule**⟨*LoadRs*⟩, which dequeues the load response and updates the appropriate architectural register with the value obtained from the load response.

On a store instruction, the processor executes **Rule**⟨*StoreRq*⟩ followed by **Rule**⟨*StoreRs*⟩. **Rule**⟨*StoreRq*⟩ and **Rule**⟨*StoreRs*⟩ behave similarly to **Rule**⟨*LoadRq*⟩ and **Rule**⟨*LoadRs*⟩, respectively, in sending an appropriate memory request and waiting for the appropriate response before changing the program counter.

### 3.2.2 Buffers between Processors and Memory

We will now describe the module $M_{\text{wrap}}$ that separates the decoupled processors from the instantaneous memory. As discussed earlier, this module contains the buffers used in such a separation. There are two buffers *cRqs*, which allows reading any element from the buffer; and $toC$, which is a FIFO buffer.

The *cRqs* buffer is similar to the *FIFO* module in Figure 2-1, except that it allows

```
Module:
Regs pc(pc_0), s(s_0), w(False);

Rule NonMemory:
let i = getInst(pc);
let⟨Nm, x⟩ = dec(s, pc, i);
when(w = False) ⇒
    let⟨s', cpc⟩ = exec(s, pc, ⟨Nm, x⟩);
    s := s';
    pc := cpc;

Rule LoadRq:
let = getInst(pc);
let⟨Ld, va, r_v⟩ = dec(s, pc, i);
when(w = False) ⇒
    cRqs.enq(⟨Ld, va, ε⟩);
    w = True;

Rule LoadRs:
let i = getInst(pc);
let⟨Ld, va, r_v⟩ = dec(s, pc, i);
let⟨Ld, v⟩ = toC.pop;
when(w = True) ⇒
    let⟨s', cpc⟩ = exec(s, pc, ⟨Ld, r_v, v⟩);
    s := s';
    pc := cpc;
    w := False;

Rule StoreRq:
let i = getInst(pc);
let⟨St, va, v⟩ = dec(s, pc, i);
when(w = False) ⇒
    cRqs.enq(⟨St, va, v⟩);
    w = True;

Rule StoreRs:
let i = getInst(pc);
let⟨St, va, v⟩ = dec(s, pc, i);
when(w = True) ⇒
    let⟨s', cpc⟩ = exec(s, pc, ⟨St⟩);
    toC.pop;
    s := s';
    pc := cpc;
    w := False;

Rule Halt:
let i = getInst(pc);
let⟨Halt⟩ = dec(s, pc, i);
when(w = False) ⇒
    halt;
```

Figure 3-4: $P_{\text{dec}}$: A decoupled processor serving as specification of $P_{\text{so}}$

any element in it to be read and removed. Therefore, it has the following methods:

- *enq*$(op, va, vt)$ : It inserts a request message $\langle op, va, vt\rangle$ into the buffer and is guarded by the condition that the buffer cannot be full when it is called.

- *extract*$(n)$ : It removes the entry in position $n$ in the buffer and returns the value of that entry, in other words, extracting the element from the buffer. The returned value is a tuple $\langle op, va, vt\rangle$.

The FIFO buffer $toC$ has an interface similar to that shown in Figure 2-1. It has only the following two methods:

- *enq*$(x)$ : It inserts a response message $x$ into the buffer where $x$ can either be $\langle \mathsf{Ld}, v\rangle$ or $\langle \mathsf{St}\rangle$. This method is guarded by the condition that the buffer cannot be full when it is called.

- *pop* : It returns the first element of the buffer which can either be $\langle \mathsf{Ld}, v\rangle$ or $\langle \mathsf{St}\rangle$. This method is guarded by the condition that the buffer cannot be empty when it is called.

In addition to the two buffers, the $M_{\text{wrap}}$ module also has rules for processing load requests and store requests, respectively. These are contained inside another module $M'_{\text{wrap}}$ as shown in Figure 3-5. Note that the entries in the *cRqs* buffer can be accessed non-deterministically. Hence there is a rule for accessing each entry in the *cRqs* buffer.

Finally, the $M_{\text{wrap}}$ module composed of the $M'_{\text{wrap}}$ module and the *cRqs* and $toC$ buffers is shown below.

**Definition 2.** $M_{\text{wrap}} \triangleq M'_{\text{wrap}} + cRqs + toC$

The combination of the $M_{\text{wrap}}$ module together with the $P_{\text{dec}}$ processor behaves exactly similar to the instantaneous processor $P_{\text{inst}}$. We will formally prove this in Theorem 11, after formally defining what is meant to behave in the same manner in Chapter 6. The intuition behind this is as follows. Whenever the $P_{\text{dec}}$ processor

```
Module:
Rule LdMem_1:
let⟨Ld, a, t⟩ = cRqs.extract(1);
toC.enq(⟨Ld, t, M_inst.ldRq(a)⟩);
...
Rule LdMem_m:
let⟨Ld, a, t⟩ = cRqs.extract(m);
toC.enq(⟨Ld, t, M_inst.ldRq(a)⟩);

Rule StMem_1:
let⟨St, a, v⟩ = cRqs.extract(1);
M_inst.stRq(a, v);
toC.enq(⟨St⟩);
...
Rule StMem_m:
let⟨St, a, v⟩ = cRqs.extract(m);
M_inst.stRq(a, v);
toC.enq(⟨St⟩);
```

Figure 3-5: Module $M'_{\text{wrap}}$: Rules for processing load and store requests from the buffers

executes a non-load or a non-store instruction, then the state changes that it undergoes mimics exactly what happens in the $P_{\text{inst}}$ processor for executing the same instruction. When $P_{\text{dec}}$ executes a memory instruction, it first sends a request to the memory system and goes into a wait state which stalls any execution till a response from the memory is available. Once the response is available, then it undergoes the same state changes as $P_{\text{inst}}$ would have underwent for executing the same memory instruction. So, in effect, $P_{\text{dec}}$ follows the same state transitions as $P_{\text{inst}}$, thus behaving in the same manner.

One thing to note in the above discussion is that the combination of $P_{\text{dec}}$ and $M_{\text{wrap}}$ behaves like $P_{\text{inst}}$ independent of the memory that the processors are connected to, as long as the memory behaves in an identical manner in both the systems. This is the key property that enables us to do modular reasoning in the presence of modular refinement. Given a system containing two modules $A$ and $B$, when we replace $A$ with another module $A'$ and prove that $A'$ behaves just like $A$, this gives us a strong compositional reasoning property enabling us to conclude that $A' + B$ behaves just like $A + B$. In fact, most hardware designs are done using modular refinement, where the designer informally adheres to compositional reasoning.

### 3.2.3 Atomic Memory: Instantaneous Memory + Decoupling Buffers

We now describe the atomic memory $M_{\text{atomic}}$ which is formed by composing the instantaneous memory $M_{\text{inst}}$ with several $M_{\text{wrap}}$ modules. The $M_{\text{wrap}}$ modules provide the interface to communicate with the processor and thus should be replicated as many times as the number of processors in the system.

**Definition 3.** $M_{\text{atomic}} \triangleq M^1_{\text{wrap}} + M^2_{\text{wrap}} + \ldots + M^n_{\text{wrap}} + M_{\text{inst}}$

### 3.2.4 Decoupled Multiprocessor System

We are now in a position to give the design of a decoupled multiprocessor system composed of the decoupled processors connected to an atomic memory. This is shown below. We will call this composition $SC_{\text{dec}}$.

**Definition 4.** $SC_{\text{dec}} \triangleq P^1_{\text{dec}} + P^2_{\text{dec}} + \ldots + P^n_{\text{dec}} + M_{\text{atomic}}$

As discussed briefly earlier, the combination of $P_{\text{dec}}$ and $M_{\text{wrap}}$ behaves just like the $P_{\text{inst}}$ processor. And compositional reasoning allows us to conclude that when several of the combination of $P_{\text{dec}}$ and $M_{\text{wrap}}$ modules are connected to the instantaneous memory $M_{\text{inst}}$, it will behave exactly like the composition of several of $P_{\text{inst}}$ modules with $M_{\text{inst}}$. This compositional reasoning property will be formalized for the general case in Part II of the thesis.

The $M_{\text{atomic}}$ module constitutes the specification what is called the memory subsystem in modern multiprocessors, while the $P_{\text{dec}}$ module constitutes the specification of the processor or core subsystem. Any realistic multiprocessor system employs several optimizations in the design of both the processor and the memory subsystem. We will discuss an optimized implementation of the processor subsystem in the form of a speculative out-of-order processor in the next section and an optimized implementation of the memory subsystem in the form of a hierarchy of coherent caches in the next chapter.

## 3.3 Speculative Out-of-Order Processor

We will now show the design of a *speculative out-of-order* processor module $P_{so}$, which (a) speculatively fetches instructions without having resolved the branches (and hence without knowing the next program counter), (b) executes instructions out of order, and (c) creates many simultaneous outstanding (speculative) load requests to the memory. These features increase the performance by doing potentially useful work whenever the processor would have previously been idle. Instructions may have long execution latencies, which are potentially hidden by speculatively doing future work. Moreover, this processor, in a real system, would be connected to a memory subsystem, where memory operations can potentially have long latencies, which are again hidden by speculative and out-of-order execution.

### 3.3.1 Components of the Speculative Out-of-Order Processor

The speculative processor consists of three component modules: (1) $P'_{so}$ which consists of the "architectural state" of the processor and rules that operate on this architectural state, (2) a *branch predictor bp*, which makes guesses about the program counter to fetch the next instruction from (in advance of executing the previous instruction to resolve branches;) and (3) a *reorder buffer rob*, which decides what (speculative) instructions are issued at which moments, enabling out-of-order execution of instructions. The speculative out-of-order processor is *abstracted* over the implementations of the branch predictor (*bp*) and reorder buffer (*rob*) modules.

#### Branch Predictor

The branch predictor module *bp* has the following methods:

- *currPc* returns *pc* where *pc* is the current program counter prediction.
- *nextPc* advances the branch predictor to predicting the next instruction's program counter.

- *setCurrPc*($pc$) resets the branch predictor's prediction to begin at position $pc$. This method is called in the case of a misprediction.

- *trainBp*($ppc, pc$) is used to train the branch predictor, by associating $pc$ to be the next instruction to execute after $ppc$. This is called whenever an instruction finishes or aborts and thus has the next instruction's program counter resolved.

We need to impose no explicit conditions for correctness of the branch predictor; the processor uses predictions only as hints, and it always resets the predictor using *setCurrPc* after detecting a misprediction.

### Reorder Buffer

The reorder buffer module *rob* has the following methods. As can be seen, the interface of the reorder buffer is more complicated than the branch predictor's.

- *add*($pc, i, ppc$), which appends the program instruction $i$ at location $pc$ to the list of instructions that are available for execution in the reorder buffer. The argument $ppc$ is the previous instruction's program counter, to be supplied to the branch predictor.

- *getLd*, which returns either a load address to be speculatively issued to the memory, along with the reorder buffer's tag for the corresponding load instruction or $\epsilon$ if the reorder buffer does not wish to issue a load. (The reorder buffer's tag is used to identify a particular instruction in the reorder buffer.)

- *sentLd*($t$), which informs the reorder buffer that the load corresponding to the instruction associated with tag $t$ has been speculatively issued.

- *ldRs*($t, v$), which informs the reorder buffer that the memory has returned result value $v$ for the speculative load with tag $t \neq \epsilon$.

- *oldest*, which returns the resulting state of executing the next instruction in serial program order, if the reorder buffer has performed enough computation

steps and gathered the required speculative load responses to execute it accurately; otherwise it returns $\epsilon$. When *oldest* returns a non-$\epsilon$ value, it returns a tuple $\langle s, pc, cpc, ppc, x\rangle$, where $s$ is the updated state on executing the next instruction, $pc$ is the program counter of that instruction, $cpc$ is the next program counter we would advance to afterward and $ppc$ is the program counter for the previous instruction. $x$ is $\epsilon$ for non-load instructions; $\langle \mathsf{Ld}, va, r_v, v\rangle$ for load instructions, where $va$ denotes the load address, $v$ denotes the speculative load response obtained from the memory, and $r_v$ denotes the register that the load value is written into; $\langle \mathsf{St}, va, v\rangle$ for store instructions where $va$ and $v$ denote the word address and value of the store, respectively; or $\langle \mathsf{Halt}\rangle$ for halt instruction. We will see why we need this information for loads shortly.

- *commit*, which informs the buffer that its *oldest* instruction was executed successfully, *i.e.,* committed, so it is time to move on to the next instruction.

The reorder buffer has internal rules to execute each instruction. It breaks down an instruction into smaller micro-ops and executes them one-by-one. Executing these micro-ops is effectively equivalent to invoking the *dec* and *exec* functions to obtain the next program counter, register values, *etc.* It keeps track of the data dependency between the instructions and tries to execute instructions out of order as long as the data dependencies are met.

We do not give the details of the reorder buffer module (like the internal state or the rules of the reorder buffer). Instead, we specify an invariant that the reorder buffer should obey.

The invariant establishes a relationship between the updates to the architectural registers and program counter returned by method *oldest* of the reorder buffer, and the instruction corresponding to the current program counter. This relationship is given precisely by the *getInst*, *dec* and *exec* functions as shown in Invariant 1. The architectural state and program counter updates that the reorder buffer returns on calling its method *oldest* should match the values obtained applying the *getInst*, *dec* and *exec* functions on the current architectural state and program counter (and the

response from memory in case of a load).

**Invariant 1.** If $s$ and $pc$ are the current values of the architectural registers and program counter, respectively, and if $\langle s', _, cpc, _, x\rangle = rob.oldest$, then

$$\begin{aligned}
x = \epsilon &\Rightarrow \exists y.\ dec(s, pc, getInst(pc)) = \langle \mathsf{Nm}, y\rangle \wedge \\
&\qquad exec(s, pc, \langle \mathsf{Nm}, y\rangle) = \langle s', cpc\rangle \\
x = \langle \mathsf{Ld}, va, r_v, v\rangle &\Rightarrow dec(s, pc, getInst(pc)) = \langle \mathsf{Ld}, va, r_v\rangle \wedge \\
&\qquad exec(s, pc, \langle \mathsf{Ld}, r_v, v\rangle) = \langle s', cpc\rangle \\
x = \langle \mathsf{St}, va, v\rangle &\Rightarrow dec(s, pc, getInst(pc)) = \langle \mathsf{St}, va, v\rangle \wedge \\
&\qquad exec(s, pc, \langle \mathsf{St}\rangle) = \langle s', cpc\rangle \\
x = \langle \mathsf{Halt}\rangle &\Rightarrow dec(s, pc, getInst(pc)) = \langle \mathsf{Halt}\rangle
\end{aligned}$$

In this sense we are not going to give a complete description of the speculative out-of-order processor; we are *abstracting* the reorder buffer component while giving its specification in terms of an invariant.

### The $P'_{so}$ module

The $P'_{so}$ component of the processor has the following registers: (a) $pc$, which gives the program counter of the next instruction that the reorder buffer has to commit, (b) $s$, which is the register array, storing the value of each *architectural register*, and (c) $w$, which denotes whether the processor is waiting for a memory response. The registers $pc, s$ and $w$ are initialized to $pc_0, s_0$ and False respectively. Note that the internal rules of the reorder buffer cannot update any of these registers.

The processor makes calls *cRqs.enq* to send requests to the memory and receives memory responses via calls to *toC.pop*. It also makes calls to *halt* on executing a halt instruction.

Figure 3-6 gives the rules in the processor $P'_{so}$. While the processor fetches arbitrary instructions speculatively in **Rule**⟨*Fetch*⟩, the rest of the rules ensure that the processor only *commits*, *i.e.,* makes changes to the architectural registers, in serial program order. The function $getInst(pc)$ returns the instruction present in the lo-

```
Module:
Regs pc(pc_0), s(s_0), w(False);

Rule Fetch:
let⟨predPc, ppc⟩ = bp.currPc;
rob.add(predPc, getInst(predPc), ppc);
bp.nextPc;

Rule SpecLoadRq:
let x = rob.getLd;
when(x ≠ ε) ⇒
    let⟨va, t⟩ = x;
    rob.sentLd(t);
    cRqs.enq(⟨Ld, va, t⟩);

Rule SpecLoadRs:
let⟨Ld, t, v⟩ = toC.pop;
when(t ≠ ε) ⇒
    rob.ldRs(t, v);

Rule Abort:
let⟨s', cpc, npc, ppc, _⟩ = rob.oldest;
when(w = False ∧ cpc ≠ pc) ⇒
    bp.trainBp(ppc, pc);
    rob.setEmpty;
    bp.setCurrPc(pc);

Rule NonMemory:
let⟨s', cpc, npc, ppc, ε⟩ = rob.oldest;
when(w = False ∧ cpc = pc) ⇒
    bp.trainBp(ppc, pc);
    rob.commit;
    s := s';
    pc := npc;

Rule Halt:
let⟨s', cpc, npc, ppc, ⟨Halt⟩⟩ = rob.oldest;
when(w = False ∧ cpc = pc) ⇒
    bp.trainBp(ppc, pc);
    rob.commit;
    halt;

Rule LoadRq:
let⟨s', cpc, npc, ppc, ⟨Ld, va, r_v, v⟩⟩ = rob.oldest;
when(w = False ∧ cpc = pc) ⇒
    cRqs.enq(⟨Ld, va, ε⟩);
    w = True;

Rule LoadRs:
let⟨Ld, t, v'⟩ = toC.pop;
let⟨s', cpc, npc, ppc, ⟨Ld, va, r_v, v⟩⟩ = rob.oldest;
when(w = True ∧ cpc = pc ∧ t = ε ∧ v = v') ⇒
    bp.trainBp(ppc, pc);
    rob.commit;
    s := s';
    pc := npc;
    w := False;

Rule LoadRsBad:
let⟨Ld, t, v'⟩ = toC.pop;
let⟨s', cpc, npc, ppc, ⟨Ld, va, r_v, v⟩⟩ = rob.oldest;
when(w = True ∧ cpc = pc ∧ t = ε ∧ v ≠ v') ⇒
    bp.trainBp(ppc, pc);
    let⟨s'', npc'⟩ = exec(s, pc, ⟨Ld, r_v, v⟩);
    s := s'';
    pc := npc';
    rob.setEmpty;
    w := False;

Rule StoreRq:
let⟨s', cpc, npc, ppc, ⟨St, va, v⟩⟩ = rob.oldest;
when(w = False ∧ cpc = pc) ⇒
    cRqs.enq(⟨St, va, v⟩);
    w = True;

Rule StoreRs:
let⟨s', cpc, npc, ppc, ⟨St, va, v⟩⟩ = rob.oldest;
when(w = True ∧ cpc = pc) ⇒
    bp.trainBp(ppc, pc);
    toC.pop;
    rob.commit;
    s := s';
    pc := npc;
    w := False;
```

Figure 3-6: Module $P'_{\text{so}}$: The speculative out-of-order processor module (the reorder buffer and the branch predictor are separate modules)

cation pointed to by the program counter *pc*. While in a typical processor this will involve an access to the memory, we will assume that the instructions are directly accessible from within the processor via the function *getInst*. The fetched instructions are inserted into the reorder buffer.

If the reorder buffer has to execute a load instruction, after computing its address, it speculatively issues the load request to memory in **Rule**⟨*SpecLoadRq*⟩. It gets a response for such a speculatively issued load in **Rule**⟨*SpecLoadRs*⟩. A speculatively issued load request has a tag $t \neq \epsilon$ identifying the instruction corresponding to the request, and this tag is returned in the response from the memory.

In case the instruction to be committed next has been fetched because of a misprediction, the processor discards this instruction and every instruction fetched after that from the reorder buffer in **Rule**⟨*Abort*⟩. The processor figures out if a misprediction has occurred by comparing the program counter of the next instruction to commit with that of the actual program counter (which was set by the previously committed instruction); they do not match in case of a misprediction. The branch predictor is reset to start fetching instructions from the correct program counter, and the correction for the prediction is supplied for the branch predictor.

If the instruction to be committed has not been mispredicted, then the processor commits that instruction. The processor updates the architectural registers and the next program counter as directed by the rest of the information that the reorder buffer supplies for committing an instruction. This finishes executing a non-memory instruction as seen in **Rule**⟨*NonMemory*⟩. For a halt instruction, it calls the *halt* method in addition, as seen in **Rule**⟨*Halt*⟩.

For committing a store instruction, the processor issues a store request and waits for a response. The reorder buffer gives the store address and value for the next instruction to commit, and the processor issues a store request using this information in **Rule**⟨*StoreRq*⟩. Finally the store response is obtained in **Rule**⟨*StoreRs*⟩.

**Need for Verification Loads:** Committing a load instruction is more complicated. Since a load for that instruction has already been performed speculatively, one maybe

tempted to simply commit the instruction and change the state of the architectural registers appropriately. This is in line with how non-load and non-store instructions are treated: at the time of commit, they just change the state of the architectural registers. However, this is incorrect even in the case of single processors. Consider the case when the processor has to execute the following program (we are showing the decoded instructions for readability):

| $P$ |
|---|
| St $va$ $v$ |
| Ld $va$ $r_v$ |

In the above example, there is a store instruction followed by a load instruction to the same address. If the load was speculatively executed before the store, then it would have received the old value of address $va$ from the memory. Then the store instruction would have stored the value $v$ into address $va$ when it is committed. When the load is committing, it will update register $r_v$ with the old value of address $va$ (since it does not read the latest value $v$ from the memory at the time of commit), violating sequential consistency even in the presence of a single processor.

One can come up with clever schemes to prevent speculative execution of load instructions whenever there is an earlier store instruction that has not been committed. This would certainly avoid the above problem in single processor systems. But in the presence of multiple processors, this solution is not sufficient. Consider a two processor system executing the following multithreaded program in processors $P_1$ and $P_2$.

| $P_1$ | $P_2$ |
|---|---|
| St $va_1$ 1 | Ld $va_2$ $r_2$ |
| St $va_2$ 1 | Ld $va_1$ $r_1$ |

The first processor executes two store instructions, both storing the value 1 into addresses $va_1$ and $va_2$, in order. The second processor executes two load instructions,

reading addresses $va_2$ and $va_1$, in order. Let us say the initial values in the memory for addresses $va_1$ and $va_2$ are both 0. If the sequence of rule execution is as follows:

1. $P_1$ fetches (St $va_1$ 1).
2. $P_1$ fetches (St $va_2$ 1).
3. $P_2$ fetches (Ld $va_2$ $r_2$).
4. $P_2$ fetches (Ld $va_1$ $r_1$).
5. $P_2$ speculatively sends load request for (Ld $va_1$ $r_1$).
6. Memory returns value 0 for address $va_1$.
7. $P_2$ receives load response for (Ld $va_1$ $r_1$), which must return value 0.
8. $P_1$ sends store commit request for (St $va_1$ 1).
9. Memory updates the value for address $va_1$ to 1.
10. $P_1$ receives store commit response for (St $va_1$ 1).
11. $P_1$ sends store commit request for (St $va_2$ 1).
12. Memory updates the value for address $va_2$ to 1.
13. $P_1$ receives store commit response for (St $va_2$ 1).
14. $P_2$ speculatively sends load request for (Ld $va_2$ $r_2$).
15. Memory returns value 1 for address $va_2$.
16. $P_2$ receives load response for (Ld $va_2$ $r_2$), which must return value 1.
17. $P_2$ commits (Ld $va_2$ $r_2$), updating register $r_2$ with value 1.
18. $P_2$ commits (Ld $va_1$ $r_1$), updating register $r_1$ with value 0.

At the end of the above execution sequence, $r_1$ and $r_2$ have the values 0 and 1, respectively. However, in any sequentially consistent two processor system, this state can never be reached at the end of executing the above program. This is because, whenever (Ld $va_2$ $r_2$) loads a value 1 both the stores in $P_1$ must have completed, thereby forcing (Ld $va_1$ $r_1$) to also read the value 1.

In order to prevent such an incorrect behavior in a multiprocessor setting, at the time of committing a load instruction, a processor issues another load request, called the *verification load* in **Rule**$\langle$*LoadRq*$\rangle$, and waits for a response. Notice that this is a second load request issued for every load instruction; the previous request was issued speculatively. The tag sent for the verification load is $\epsilon$ as there is only one outstanding verification load and the processor waits for its response before proceeding further. On receiving the response for the verification load, if the load value matches the value obtained as a response to a previously issued speculative load for the same instruction, then the processor commits that instruction, switching out of the wait state, as seen in **Rule**$\langle$*LoadRs*$\rangle$. On the other hand, if the verification load's response does not match what was previously obtained, then the new value is used to re-execute the load instruction, committing the result of the re-execution. A mismatch between the verification load response and the speculative load response for a load instruction is treated like a branch misprediction; the rest of the instructions in the reorder buffer are discarded and the branch predictor is reset to start fetching from the next instruction's program counter obtained from the re-execution of the load instruction. This is seen in **Rule**$\langle$*LoadRsBad*$\rangle$.

In common cases, there is no loss in performance because of executing loads twice, as it is likely that the verification load finds the address already in a local cache, thanks to the recent processing of the speculative load. Moreover, 60% to 90% of verification loads can be avoided by invalidating speculative loads [16] earlier.

The full processor $P_{\text{so}}$ is composed of $P'_{\text{so}}$, the reorder buffer and the branch predictor:

**Definition 5.** $P_{\text{so}} \triangleq P'_{\text{so}} + bp + rob$

## 3.4 Full-system Verification of the Multiprocessor

As discussed earlier, the implementations of the memory subsystem and the processor subsystem are very different and complex compared to their specifications. But, in spite of their complexities, these optimized out-of-order cores and cache hierarchy should fundamentally just implement their specifications. Moreover, the decoupled system specification consisting of several decoupled processors composed with the atomic memory specification should itself implement the sequential consistency specification of Section 3.1.

In order to prove that a system $A$ implements a specification $S$, we need to first formalize the notion of "implements". Formalizing this notion and proving that an implementation meets the specification is the subject of the second part of this dissertation. In this section we will briefly describe some of the difficulties in formalizing the notion of implements for the processor example, *i.e.*, for proving that $P_{\text{so}}$ implements $P_{\text{dec}}$.

An intuitive way of formalizing the notion of "implements" is to ensure that the interaction which an implementation has with any external environment is the same as what the specification has. In other words, in any context, the implementation is indistinguishable from the specification.

Unfortunately, this simple a notion of "implements" does not work very well for our processor example. Notice that the $P_{\text{so}}$ processor issues two kinds of loads to the memory subsystem – the speculative load and the "real" load at the time of commit, while the $P_{\text{dec}}$ processor issues only one kind of load to the memory, *viz.*, the real load. Clearly, the interactions with the external environment are different.

However, if the notion of implements is relaxed enough to not require the interactions to exactly match, and in particular, if we are allowed to "omit" the speculative load interactions (both requests and responses), then $P_{\text{so}}$ can be verified to be indistinguishable from $P_{\text{dec}}$ with respect to the rest of the interactions. Chapter 6 shows exactly how to define such an implements relation, and Chapter 7 shows how to verify the $P_{\text{so}}$ processor using this relaxed definition.

It is worth noting that if we stretch this argument, we can omit all interactions and prove that any system implements any other system. Fortunately, such absurd results can be ruled out by noting that we are interested in proving that a composition of several modules implements another system, perhaps a composition of another set of modules. The interactions between the modules in this composition do not go away, even if we relax our definition of implements. So, unless the relaxed definition is useful to allow interesting compositions while still being able to prove full-system properties, using a relaxed definition does not work.

## 3.5 Conclusion

In this chapter we first gave the sequential consistency specification module. We then presented a simple decoupled system consisting of decoupled processors connected to an atomic memory, where the processors and memory communicate only via buffers. We then presented the speculative out-of-order processor, which is an optimized implementation of a decoupled processor. In the next chapter, we will discuss the design of a hierarchy of caches implementing a directory-based cache coherence protocol, which is an optimized implementation of the atomic memory specification. In Part II, we will prove that each of these complex designs actually implements their respective specifications, and the overall system composed of these complex components implements sequential consistency.

# Chapter 4

# Implementing the Memory Subsystem Using Coherent Caches

In this chapter we will design a hierarchy of coherent caches which implements the atomic memory specification defined in Section 3.2.3. This forms the memory subsystem of any modern processor. The cache-coherence protocol used is a directory-based invalidation protocol.

We will first describe the high-level structure of the hierarchy of caches in Section 4.1. We will then give detailed designs of the cache nodes in Bluespec in Section 4.2. The specification of each cache node specifies the cache-coherence protocol completely, which is essentially a distributed algorithm where caches send messages to each other to keep each of them "coherent". The caches send various kinds of messages, all of which will be specified in detail in Section 4.2. There are certain constraints on the interaction between various kinds of messages – constraints of the form "a response from a child to its parent should never be blocked by a request from that child to the parent in order to satisfy the protocol". We will give all such constraints in Section 4.3.

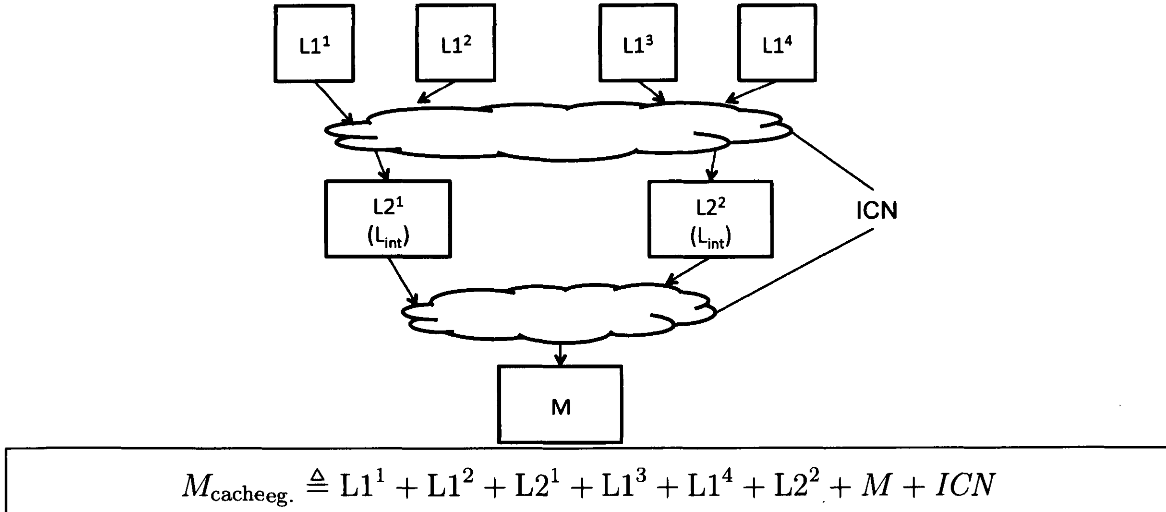



Figure 4-1: Example instantiation of the memory subsystem

## 4.1 Structure of Hierarchy of Caches

The memory subsystem $M_{\text{cache}}$ consists of a hierarchy of coherent caches organized as a logical tree, connected by an interconnect network *ICN*. Figure 4-1 shows an instance of such a system.

While the cache nodes are organized logically as a tree, the physical topology in which they are organized can be very different. As shown in Figure 4-1, all these caches communicate with each other through a single physical interconnect network. There are various topologies that are widely used for implementing this network, for instance ring [20, 40, 70, 79, 85], torus [2, 28], mesh [46, 73, 84, 88], hypercube [29, 39] *etc.*. (Appendix F: Interconnection Networks in Hennessy *et al.* [41] gives an overview of various network topologies, and Dally *et al.* [27] gives a comprehensive overview of the same.) In fact, in a real implementation, usually even a single cache node that we describe in Section 4.2 is physically distributed – a cache node is partitioned based on address, and the different partitions corresponding to different address spaces usually resides in different locations in a physical chip [8].

Throughout this chapter, we will abstract the actual topology of the physical network. We will also abstract the partitioning of a cache node based on address space and distributing them physically – our specification of the cache nodes is agnostic to

address space partitioning. The only requirement for the physical network is that they satisfy all the constraints on message interactions that we specify in Section 4.3 – these constraints can be implemented using a single physical network via multiple *virtual channels* or *virtual networks* [81].

## 4.2 Cache Nodes

There are three kinds of cache nodes in this system:

1. the leaf nodes, also called the L1 caches (module L1), which are the ones connected to the processors;

2. the internal caches (module $L_{int}$); and

3. the root node, also called the memory (module $M$).

While the L1 and internal caches are truly caches, in the sense that they cache only the values of a (proper) subset of addresses in them, the memory has enough storage to keep the values of every address in it. (But as can be seen in Section 4.2.4, even though the memory has space to keep the values for every address, the values it keeps may not be valid or up-to-date.)

The behavior of the three kinds of caches are very similar. The internal cache can be considered as the canonical cache and the other caches are minor variants of this. The memory $M$ is just an internal cache without any parent, and hence it does not contain the structures and behaviors which are involved with interacting with the parent. The L1 caches is just an internal caches except that they interact with processors instead of with children, and change their structures and behaviors appropriately. We will see all of these caches in greater detail below.

The register arrays inside a cache node is shown in Figure 4-2. The L1 caches have *cs*, *w*, *tags* and *data* register arrays, for storing the MSI cache state [41], the boolean wait state, the address tags and the data, respectively and the memory has *dir*, *dirw* and *data* arrays, for storing the directory state, the boolean directory-wait state and

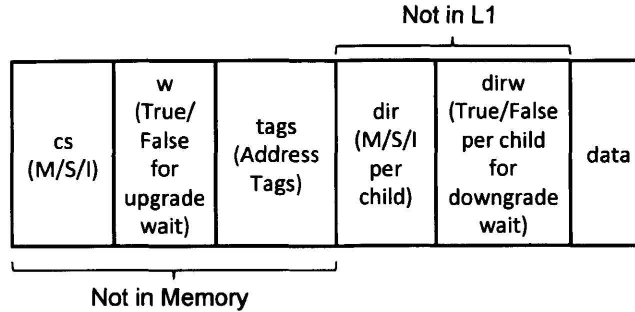



Figure 4-2: Register Arrays in each cache node

the data, respectively. The internal caches have the union of these two sets of register arrays.

For precision, we will describe each of these nodes in great detail by giving their transition rules in the sections below.

The cache coherence protocol that we are designing is an *invalidation protocol* based on MSI [41] which ensures that whenever an L1 cache has an address in $M$ state (*i.e.,* the corresponding line is writable by the processor attached to the L1 cache), no other L1 cache has the same address in $S$ state, where the corresponding line is readable by the attached processor, or in $M$ state. If an L1 cache has an address in $M$ or $S$ state when a processor attached to another L1 cache wants to write to that address, then the former cache *invalidates* the cache line corresponding to the address, changing the cache state of the line to $I$. This is in contrast to the *update protocol* [38] which allows other processors to read the address, but ensures that the readers are *updated* with the latest written value. We use $x > y$ to denote that cache state $x$ has more permissions than cache state $y$. Thus, $M > S > I$.

### 4.2.1 Abstractions Used in the Cache Nodes

While the specification of the cache node that we specify below are complete to implement a real cache, we do abstract some of the details. In particular, we abstract the *associativity* and the replacement policies associated with a cache node [41].

A cache only stores the lines corresponding to a subset of the complete set of addresses. Throughout this dissertation, we call the locations which store the data

and other information for a line as a *cache slot.* The associativity of a cache determines which addresses can be stored in different cache slots – a *fully associative cache* can store any address in any slot; a *direct mapped cache* has only one slot for every address; and an *n-way set associative cache* has $n$ slots for every address and can therefore store an address in any one of the $n$ slots associated with the address. All our specification is based on slots and therefore becomes independent of the actual organization of the cache.

Whenever a cache runs out of slots to store the line corresponding to a new address, it chooses one slot into which that address can be mapped as the *victim slot*, and evicts the original address present in that slot (if necessary). The algorithm or the policy to choose the victim among the set of potential victims is known as the replacement policy, and we abstract this algorithm using a function *getVictim* which will be discussed below. Usually, we need to keep track of more information in order to implement a good replacement policy. For instance, to implement a *least recently used* replacement policy, which evicts the slot which was least recently used, one needs to keep track of the timestamps when the slots were accessed. While we do not explicitly have any such state in our specifications, this state is orthogonal to the cache specification and is used only by *getVictim* function.

### 4.2.2 Naming Conventions used in the Cache Nodes

Before we delve into the design of the cache nodes, we will give an overview of the naming conventions that we use. We name the cache-id for a particular module under discussion as *cid*, and the parent of *cid* as *pid*.

Each cache has several buffers associated with it. We name them using a combination of the source or destination of the messages that the buffer carries and the type of messages that the buffer carries. We use *From* to denote buffers carrying messages "from" a source and *To* to denote buffers carrying messages "to" a destination. We use *C* to denote "child" and *P* to denote "parent". Finally, we use *rq* or *Rq* to denote "requests" and *rs* or *Rs* to denote "responses". Thus, *rqFromC* denotes a buffer which carries requests from the children of the cache under discussion while *rsToP* denotes

a buffer which carries responses to the parent of the cache under discussion. If *rq* or *rs* is omitted, it means that the buffer carries both requests and responses. For instance, $toC$ buffer carries both requests and responses to the children of the cache under consideration. Finally, for L1 caches, we overload the character $C$ to denote "core" instead of child. So $rqFromC$ in an L1 cache would mean a buffer carrying requests from a core/processor. Similarly $toC$ in an L1 cache would mean a buffer carrying messages from an L1 cache to a processor (the message happens to be only response messages, but we use the same name for uniformity among all cache types).

Another point to note about naming is the use of primed and unprimed modules in this chapter. A primed module $A'$ denotes a primitive cache module (as opposed to a module formed by composing other modules) without any buffers associated with it, while the unprimed version of the module $A$ denotes the same cache module composed with all the surrounding buffer modules.

Finally, while there are only three kinds of caches, each type of cache is "instantiated" multiple times. As discussed in Section 2.1.3, we treat each of the instances of the module by renaming all the register, method and rule names in the modules, giving them unique names. Our convention is to use superscripts for various instances of modules. Thus, we formally define the $M_{\text{cache}}$ module as follows:

**Definition 6.** $M_{\text{cache}} \triangleq \text{L1}^1 + \text{L1}^2 + \ldots + \text{L2}^1 + \text{L2}^2 + \ldots + M$

We will now give the detailed design of each kind of cache.

### 4.2.3 L1 cache

Figure 4-3a shows the components of an L1 cache, and its module specification is given in Figure 4-4.

The complete L1 cache module consisting of all the buffers is given below.

**Definition 7.** $\text{L1} \triangleq cRqs + toC + \text{L1}' + rqToP + rsToP + fromP + pRqs$

### L1 Cache state

The L1 cache has cache state $cs[l]$ that contains the MSI state for the address in each slot $l$. Initially, all cache state values are $I$.

The L1 cache also contains information about whether a line is being upgraded, for a particular slot $l$, in the array $w[l]$. Initially, all $w$ values are False.

Finally, the cache contains a *data* array for storing the cache-line data, and a *tags* array to keep the line addresses associated with the cache line. All the data and tags are uninitialized.

### Buffers in the L1 Cache

The L1 cache has the buffers described in Figure 4-3b, which also gives the format of the message entries stored in the buffers. Note that the destination IDs are stripped off the incoming messages.

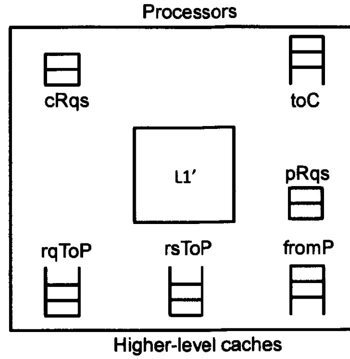
The FIFO buffers in Figure 4-3b are similar to the *FIFO* module in Figure 2-1. They can have any capacity as opposed to a capacity of two in Figure 2-1. The *toC*, *rqToP*, *rsToP* and *fromP* FIFO buffers have the following methods:

- *enq*($x$): Enqueues message $x$ into the buffer and is guarded by the buffer not being full.

- *pop*: Dequeues oldest enqueued message from the buffer and returns that message. It is guarded by the buffer not being empty.

In addition, the L1 cache also has two buffers *cRqs* and *pRqs*. These buffers can also be implemented similarly to the *FIFO* module in Figure 2-1, and these modules also will not have any rules.

The *cRqs* buffer is what is traditionally called the Miss-Status Holding Registers (MSHRs) [53,54,78,82]. We use *cRqs* to denote an unordered set of requests from the core. Similarly, we use *pRqs* to denote an unordered set of requests from the parent. The name *cRqs* is chosen to keep it consistent with the rest of the types of caches (in which *cRqs* stands for unordered set of requests from the children).




(a) Modules in an L1 cache

| Buffer | FIFO | Description | Format |
|---|---|---|---|
| *toC* | Yes | Responses to processor | ⟨Ld, load-tag, value⟩ $\mid$ ⟨St, value⟩ |
| *rqToP* | Yes | Requests to parent | ⟨parent-cache-id, current-cache-id, address, old-state, upgrade-state⟩ |
| *rsToP* | Yes | Responses to parent | ⟨parent-cache-id, current-cache-id, address, downgrade-state, data⟩ |
| *fromP* | Yes | Messages from parent | ⟨Req, address, downgrade-state⟩ $\mid$ ⟨Rs, address, upgrade-state, data⟩ |
| *cRqs* | No | Requests from processor | ⟨request-state, ⟨Ld, address, load-tag⟩⟩ $\mid$ ⟨request-state, ⟨St, address, ⟨value, store-tag⟩⟩⟩ |
| *pRqs* | No | Requests from parent | ⟨address, downgrade-state⟩ |

(b) Buffers in L1 cache, and its message formats

| $searchTags(cs, tags, a)$ | Returns a cache slot containing address $a$ in the $M$ or $S$ state, or $\epsilon$ if no such slot is found. |
|---|---|
| $searchAddr(cRqs.all, a)$ | Returns a cache slot $l$ associated with a request $\langle op, va, vt\rangle$ where $getTag(va) = a$ in the $\langle \mathsf{WaitSt}, l\rangle$ or $\langle \mathsf{WaitV}, l\rangle$ augmented state in the *cRqs* buffer, or $\epsilon$ if no such request is found. |
| $searchSlot(cRqs.all, l)$ | Returns an address $a$ associated with a request $\langle op, va, vt\rangle$ where $getTag(va) = a$ in the $\langle \mathsf{WaitSt}, l\rangle$ or $\langle \mathsf{WaitV}, l\rangle$ augmented state in the *cRqs* buffer, or $\epsilon$ if no such request is found. |
| $getVictim$ $(cRqs.all, pRqs.all, a)$ | Returns a victim cache slot for eviction to replace with address $a$ while ensuring that the victim's slot is not in use for any other request in *cRqs* buffer, and the victim's address is not present in *cRqs* and *pRqs* buffers. If no such slot is found, it returns $\epsilon$. |
| $completed(cRqs.all, x, a)$ | Returns if there any "completed" requests in *cRqs* buffer, *i.e.*, a request $\langle op, va, vt\rangle$ where $getTag(va) = a$ in augmented state $\langle \mathsf{WaitSt}, _\rangle$ and either $op = \mathsf{Ld}$ and $x \geq S$ or $op = \mathsf{St}$ and $x = M$. |

(c) Helper functions in L1 cache transitions

Figure 4-3: L1 cache

```
Module:
Regs cs[NUM]({I}), w[NUM]({False}), data[NUM]({_}), tags[NUM]({_});

Rule LdHit:
let n = cRqs.get(cs, tags, w, pRqs.all);
let⟨Init, ⟨Ld, va, t⟩⟩ = cRqs.extract(n);
let a = getTag(va);
let l = searchTags(cs, tags, a);
when((l ≠ ε) ∧ (searchSlot(cRqs.all, l) = ε)
        ∧ (searchAddr(cRqs.all, a) = ε)
        ∧ (searchAddr(pRqs.all, a) = ε)) ⇒
    toC.enq(⟨Ld, t, data[l][getOffset(va)]⟩);

Rule StHit:
let n = cRqs.get(cs, tags, w, pRqs.all);
let⟨Init, ⟨St, va, v⟩⟩ = cRqs.extract(n);
let a = getTag(va);
let l = searchTags(cs, tags, a);
when((l ≠ ε) ∧ (cs[l] = M)
        ∧ (searchSlot(cRqs.all, l) = ε)
        ∧ (searchAddr(cRqs.all, a) = ε)
        ∧ (searchAddr(pRqs.all, a) = ε)) ⇒
    toC.enq(⟨St⟩);
    data[l][getOffset(va)] := v;

Rule MissByState:
let n = cRqs.get(cs, tags, w, pRqs.all);
let⟨Init, ⟨r, va, vt⟩⟩ = cRqs.read(n);
let a = getTag(va);
let l = searchTags(cs, tags, a);
when((l ≠ ε) ∧ (r = St) ∧ (cs[l] < M)
        ∧ (searchSlot(cRqs.all, l) = ε)
        ∧ (searchAddr(cRqs.all, a) = ε)
        ∧ (searchAddr(pRqs.all, a) = ε)) ⇒
    cRqs.upd(n, ⟨WaitSt, l⟩);

Rule MissByLine:
let n = cRqs.get(cs, tags, w, pRqs.all);
let⟨Init, ⟨r, va, vt⟩⟩ = cRqs.read(n);
let a = getTag(va);
let l = getVictim(cRqs.all, pRqs.all, a);
when((l ≠ ε) ∧ (searchTags(cs, tags, a) = ε)
        ∧ (searchAddr(cRqs.all, a) = ε)
        ∧ (searchAddr(pRqs.all, a) = ε)) ⇒
    cRqs.upd(n, ⟨WaitV, l⟩);

Rule Writeback:
let n = cRqs.get(cs, tags, w, pRqs.all);
let⟨⟨WaitV, l⟩, ⟨r, va, vt⟩⟩ = cRqs.read(n);
let a' = tags[l]
if(cs[l] ≠ I) rsToP.enq(⟨pid, cid, a', I,
        ⟨if (cs[l] = M) then data[l] else _⟩⟩);
cs[l] := I;
cRqs.upd(n, ⟨WaitSt, l⟩);

Rule UpgRq:
let n = cRqs.get(cs, tags, w, pRqs.all);
let⟨⟨WaitSt, l⟩, ⟨op, va, t⟩⟩ = cRqs.read(n);
let a = getTag(va);
let x = if (op = Ld) then S else M;
when(¬w[l] ∧ (cs[l] < x)) ⇒
    rqToP.enq(⟨pid, ⟨cid, a, cs[l], x⟩⟩);
    w[l] := True;

Rule UpgRs:
let⟨Rs, va, x, v⟩ = fromP.pop;
let a = getTag(va);
let l = searchAddr(cRqs.all, a);
when(searchAddr(pRqs.all, a) = ε) ⇒
    if(cs[l] = I) then{data[l] := v; tags[l] := a; }
    cs[l] := x;
    w[l] := False;

Rule LdDeferred:
let n = cRqs.get(cs, tags, w, pRqs.all);
let⟨⟨WaitSt, l⟩, ⟨Ld, va, t⟩⟩ = cRqs.extract(n);
let a = getTag(va);
when(cs[l] ≥ S) ⇒
    cRqs.upd(n, Free);
    toC.enq(⟨Ld, t, data[l][getOffset(a)]⟩);

Rule StDeferred:
let n = cRqs.get(cs, tags, w, pRqs.all);
let⟨⟨WaitSt, l⟩, ⟨St, va, v⟩⟩ = cRqs.extract(n);
let a = getTag(va);
when(cs[l] = M) ⇒
    cRqs.upd(n, Free);
    toC.enq(⟨St⟩);
    data[l][getOffset(a)] := v;

Rule PTransfer:
let⟨Req, a, x⟩ = fromP.pop;
pRqs.ins(⟨a, x⟩);

Rule Drop:
let n = pRqs.get(cs, tags, w, cRqs.all);
let⟨a, x⟩ = pRqs.read(n);
let l = searchTags(cs, tags, a);
when((l = ε) ∨ (cs[l] ≤ x)) ⇒
    pRqs.remove(n);

Rule PProcess:
let n = pRqs.get(cs, tags, w, cRqs.all);
let⟨a, x⟩ = pRqs.read(n);
let l = searchTags(cs, tags, a);
when((cs[l] > x)
        ∧ ¬completed(cRqs.all, cs[l], a)) ⇒
    pRqs.remove(n);
    rsToP.enq(⟨pid, ⟨cid, a, x,
            if (cs[l] = M) then data[l] else _⟩⟩);
    cs[l] := x;
```

Figure 4-4: Module L1': Module containing the transition rules of the L1 cache

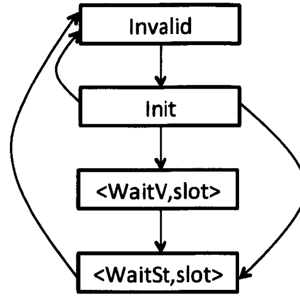



Figure 4-5: The augmented state changes for request entries in *cRqs*

The requests in a *cRqs* buffer go through several stages during the course of its processing. Every entry is augmented with information about these stages. Each entry starts in its Free augmented state (when no request is present). The entry goes into Init augmented state when it is assigned a new request. It can potentially go into waiting augmented state $\langle \mathsf{WaitV}, \mathrm{slot} \rangle$, waiting for assigning a cache line to the requesting address; or $\langle \mathsf{WaitSt}, \mathrm{slot} \rangle$, waiting for getting a response from the parent. It finally goes back to Free augmented state when the request is removed. The augmented state changes that each request entry in *cRqs* buffer undergoes are shown in Figure 4-5.

The *cRqs* buffer has the following methods:

- $ins(x)$: Inserts the request $x$ into an Free entry in the buffer, setting the augmented state of the entry to Init. It is guarded by the buffer not being full (the buffer is full if all entries are in non-Free augmented state).

- $get(cs, tags, w, pRqs.all)$: Returns a position $n$ which contains a valid (non-Free) request. It is guarded by the buffer not being empty (the buffer is empty if all entries are in Free augmented state).

- $upd(n, s)$: Updates the entry at position $n$ with augmented state $s$.

- $read(n)$: Returns a tuple $\langle s, x \rangle$ where the request $x$ is present in position $n$ in augmented state $s$.

- $extract(n)$: Returns a tuple $\langle s, x \rangle$ where the request $x$ is present in position $n$ in augmented state $s$, and sets the augmented state of position $n$ to Free thereby

removing the request in position $n$.

The requests from a *fromP* FIFO buffer enter the *pRqs* buffer, but the responses from a *fromP* FIFO buffer do not enter the *pRqs* buffer. The *pRqs* buffer has the following methods:

- *ins*($x$): Inserts request $x$ into the buffer. It is guarded by the buffer not being full.
- *get*(*cs*, *tags*, $w$, *cRqs.all*): Returns a position $n$ which contains a valid request. It is guarded by the buffer not being empty.
- *read*($n$): Returns a tuple $x$ where the request $x$ is present in position $n$.
- *remove*($n$): Removes the request in position $n$.

### Transition Rules for L1 Cache

Figure 4-4 shows the transition rules of an L1 Cache (*i.e.*, the rules of the L1′ module) for handling requests from the processor and from the parent. These rules can only directly access the local state inside the L1′ module; it calls the appropriate methods in the buffers to access them.

The first point to note that there is a distinction between a word address $va$ in the rules and the line address $a$. The processor sends requests for word addresses, from which the line address is extracted by removing the *offset*. Function *getOffset*($va$) returns the offset from a word address $va$, and function *getTag*($va$) returns the line address from a word address $va$. Function $a$ ++ $o$ gives the word address from a line address $a$ and an offset $o$, In the rest of the dissertation, an address mentioned in the context of caches is always a line address, unless explicitly stated to be a word address.

Here is an outline on how the transitions work. When an L1 cache gets a request from its processor, it checks to see if the cache state is high enough to process the request (*i.e.*, it is at least in the $S$ state for a Ld request and $M$ state for a St request). If so, it sends back a response and dequeues the request.

On receiving a request which is a miss and hence cannot be processed immediately, the L1 cache first obtains a slot to keep the address (if the address is not already in the cache). This may involve evicting another address from the cache. After obtaining a slot for the request, an upgrade request is sent to the parent. Once the response from the parent is received for the upgrade request, then the original request is dequeued and a response is sent back to the processor.

When the L1 cache receives a downgrade request from the parent it dequeues the request and sends back a response to the parent.

The transition rules use the helper functions shown in Figure 4-3c.

For notational convenience, throughout the dissertation we say that a slot $l$ contains an address $a$ if $tags[l] = a$, and a slot is in $x$ state if $cs[l] = x$.

When $getVictim(cRqs.all, pRqs.all)$ returns a non-$\epsilon$ slot, then that slot can be victimized. Address $a$ is passed as an argument because not all slots can be allotted to every address (for example, consider a directed-mapped cache).

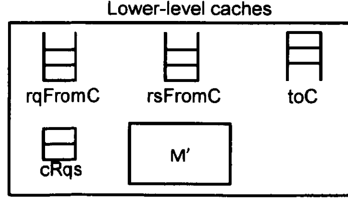
Note that we do not have dirty bits (or, as it is called in the MESI protocol [69], the $E$-state), which would have avoided sending back data in some cases when downgrading from $M$, if no stores have actually been performed after the cache obtained $M$ permissions. It is straightforward to add such an extension to this protocol.

Another point to note is that when an upgrade request is made, an association between the line address and the cache slot where the line will reside has to be maintained. This can be done either by sending the slot number with the upgrade requests, or searching the requests from the processor which are waiting to be served (as done by the *searchAddr* function). The number of additional bits needed for sending the slot number is only $\log_2$(cache-associativity).

### 4.2.4 Memory

Figure 4-6a shows the components of memory, and its module specification is given in Figure 4-7.

The complete memory module consisting of all the buffers is given below.




(a) Modules in Memory

| Buffer | FIFO | Description | Format |
|---|---|---|---|
| *rqFromC* | Yes | Requests from children | ⟨src-cache, address, old-state, upgrade-state⟩ |
| *rsFromC* | Yes | Responses from children | ⟨src-cache, address, downgrade-state, data⟩ |
| *toC* | Yes | Messages to children | ⟨dest-cache, ⟨Req, address, downgrade-state⟩⟩ \| ⟨dest-cache, ⟨Rs, address, upgrade-state, data⟩⟩ |
| *cRqs* | No | Requests from children | ⟨src-cache, address, old-state, upgrade-state⟩ |

(b) Buffers in memory, and its message formats

| | |
|---|---|
| $searchAddr(cRqs.all, a)$ | Similar to the *searchAddr* function for L1 caches. |
| $isCompatible(x, y)$ | Checks whether MSI states $x$ and $y$ are compatible. This is a commutative function. $M$ is compatible only with $I$, $S$ is compatible with both $S$ and $I$ while $I$ is compatible with any MSI state. |
| *findIncompatible* $(c, dirList, dirwList, x)$ | Returns a child cache which is not $c$, whose directory state ($dirList[c]$) is not compatible with $x$ state, and where there is no pending response from that child (*i.e.*, $\neg dirwList[c]$). |

(c) Helper functions in memory transitions

Figure 4-6: Memory

```
Module:
Regs dir[NUM]({{I}}), dirw[NUM]({{False}}), data[NUM](d0);

Rule CTransfer:
let⟨c, a, y, x⟩ = rqFromC.pop;
cRqs.ins(⟨c, a, y, x⟩);

Rule Hit:
let n = cRqs.get(dir, dirw);
let⟨Init, ⟨c, a, y, x⟩⟩ = cRqs.extract(n);
when((dir[a][c] ≤ y) ∧ (¬dirw[a][c])
        ∧ (∀i ≠ c. compat(dir[a][i], x))
        ∧ (searchAddr(cRqs.all, a) = ϵ)) ⇒
    dir[a][c] := x;
    toC.enq(⟨c, ⟨Rs, a, x,
        if (dir[a][c] = I) then data[a] else _⟩⟩);

Rule MissByState:
let n = cRqs.get(dir, dirw);
let⟨Init, ⟨c, a, y, x⟩⟩ = cRqs.read(n);
when((dir[a][c] ≤ y)
        ∧ ¬(∀i ≠ c. compat(dir[a][i], x))
        ∧ (searchAddr(cRqs.all, a) = ϵ)) ⇒
    cRqs.upd(n, WaitSt);

Rule DwnRq:
let n = cRqs.get(dir, dirw);
let⟨WaitSt, ⟨c, a, y, x⟩⟩ = cRqs.read(n);
let i = findIncompatible(c, dir[a], dirw[a], x);
when(i ≠ ϵ) ⇒
    toC.enq(⟨i, ⟨Req, a, if (x = M) then I else S⟩⟩);
    dirw[a][i] := True;

Rule DwnRs:
let⟨c, a, x, v⟩ = rsFromC.pop;
if (dir[a][c] = M) data[a] := v;
dir[a][c] := x;
dirw[a][c] := False;

Rule Deferred:
let n = cRqs.get(dir, dirw);
let⟨WaitSt, ⟨c, a, y, x⟩⟩ = cRqs.extract(n);
when((∀i ≠ c. compat(dir[a][i], x)) ∧ (¬dirw[a][c])) ⇒
    dir[a][c] := x;
    cRqs.upd(n, Free);
    toC.enq(⟨c, ⟨Rs, a, x,
        if (dir[a][c] = I) then data[a] else _⟩⟩);
```

Figure 4-7: Module $M'$: The module containing the transition rules of the memory

**Definition 8.** $M \triangleq rqFromC + cRqs + toC + M'$

### Memory state

The memory contains a directory which contains the MSI state for each line address for each child. $dir[a][c]$ gives the directory state for line address $a$ and child $c$. $dir[a]$ gives the vector of MSI states for line address $a$ for every child. Initially, all directory values are $I$.

The directory also contains information about which child $i$ is being downgraded, for a particular address $a$, in the array $dirw[a][i]$. Initially, all *dirw* values are False.

Finally, the memory contains a *data* array for storing the cache-line data, which initially contains the initial value of the memory $d_0$. This is the same as $mem_0$ in Figure 3-2 except that $m_0$ is stored in word granularity while $d_0$ is stored in line granularity. The memory does not have any *tags* array, since it stores the data for every line address and hence can be indexed directly with a line address.

### Buffers in the Memory

Figure 4-6a shows the components inside the memory. It has the buffers described in Figure 4-6b which also gives the format of the message entries stored in the buffers. The buffers in the memory are similar to the processor-side buffers in the L1 cache, and the parent-side buffers are absent since the memory has no parent. Note that the destination IDs are stripped off the incoming messages as in L1 caches.

The memory has one extra FIFO buffer *rsFromC* which carries response messages from its children. Buffers *toC*, *rqFromC* and *rsFromC* also have methods *enq* and *pop*.

The memory has a buffer *cRqs* which is similar to the buffer *cRqs* of the L1 cache. Instead of a request going through 4 stages as in the L1 cache, there are only 3 stages that a request goes through in the memory, as shown in Figure 4-8.

There are no buffers carrying messages from the parent because the memory is at the root of the memory hierarchy tree and thus has no parent.

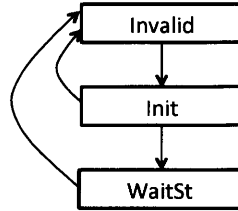



Figure 4-8: The augmented state changes for request entries in *cRqs* in the memory

### Transition Rules for Memory

Figure 4-7 shows the transition rules of memory (*i.e.,* the rules of the $M'$ module) for handling requests from the children. These rules can only directly access the local state inside the $M'$ module; it calls the appropriate methods in the buffers to access them.

Here is an outline on how the transitions work. When the memory gets an upgrade request from one of its children, it checks to see if the other children's cache states are compatible with the requesting child's upgrade. It does so by consulting its directory. The invariant that we maintain in this protocol is that the directory contains a conservative estimate of the children's cache states. If the other children's cache states are compatible, it dequeues the request, upgrades the requesting child's directory information and sends back the response.

On receiving a request which cannot be processed immediately because the other children's cache states are not compatible (as indicated by the directory), it sends downgrade requests to the non-compatible children. As and when the corresponding responses are obtained, the directory is updated to reflect the downgrades. Finally, when all the other children's cache states have become compatible, the original child's request is dequeued and responded to.

The transition rules use the helper functions shown in Figure 4-6c. They use some functions similar to the ones in L1 caches and a few others.

Requests and responses from the cache to its parent have to maintain some order, which is, a request for an address cannot overtake a response for the same address. Moreover, a request message is not allowed to block a response message. One way to solve this problem is to use the same queue for requests and responses, but prevent

request messages from blocking response messages by assigning them lower priority. We have chosen an alternative solution in which requests and responses go in separate FIFO queues, but we send some extra information (namely old cache state) and do an extra check at the parent to ensure that a future request does not overtake an earlier response. We will discuss more about this in Section 4.3.

### 4.2.5 Internal Cache

Figure 4-9a shows the components of the internal cache, and its module specification is given in Figure 4-10.

The complete internal cache module consisting of all the buffers is given below.

**Definition 9.** $\mathrm{L}_{\text{int}} \triangleq \mathit{rqFromC} + \mathit{rsFromC} + \mathit{cRqs} + \mathit{toC} + \mathrm{L}'_{\text{int}} + \mathit{rqToP} + \mathit{rsToP} + \mathit{pRqs} + \mathit{fromP}$

#### Internal Cache state

The internal cache consists of a combination of the directory and the cache states. Thus, it contains cache state $cs[l]$ for each line $l$ and directory $dir[l][c]$ for each line $l$ and child $c$. Initially the values of all these register arrays are set to $I$.

It also contains the wait-for-response array $w[l]$ and $dirw[l][c]$ to wait for responses from its parent and from child $c$ for line $l$, respectively. Initially all these register arrays are set to False.

It also contains the *data* and *tags* array, both of which are uninitialized.

#### Buffers in the Internal Cache

Figure 4-9a shows the components inside an internal cache. It has the buffers described in Figure 4-9b, which also gives the format of the message entries stored in the buffers. The FIFO buffers *rqFromC*, *rsFromC*, *toC*, *rqToP*, *rsToP* and *fromP* have methods *enq* and *pop* similar to their counterparts in the memory or the L1 cache. Note that the destination IDs are stripped off the incoming messages here also.

### Transition Rules of Internal Cache

Figure 4-10 shows the transition rules of an internal cache (*i.e.,* the rules of the $L'_{int}$ module) for handling requests from the children and from the parent, respectively. These rules can only directly access the local state inside the $L'_{int}$ module; it calls the appropriate methods in the buffers to access them.

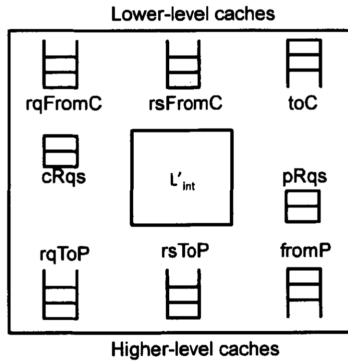
The internal cache combines the transitions rules of the L1 cache and the memory. When it gets an upgrade request from one of its children, it first checks to see if that line is present. If not, it evicts an address from another slot, similar to how the L1 cache evicts a line. However, in order to evict another address, the cache state of that address in all its children must be $I$. It consults the directory to check if this is true, otherwise it sends downgrade requests to the children to invalidate the evicted address. Once all the children have the evicted address in $I$ state, then that address can be evicted, and the slot can be alloted for the new request.

In order to service the request once a slot has been allocated, the cache has to take care of both its own cache state for the address and the cache states of its children. Its own cache state must be at least as high as the requesting upgrade, otherwise it has to send an upgrade request to its parent to upgrade its own cache state. Similarly, the cache states of the other children must be compatible, otherwise the other children have to be requested to downgrade.

Once all the responses are obtained, and hence all the children have become compatible and its own cache state is high enough, then the original request can be responded to.

The transition rules use the helper functions shown in Figure 4-9c. These are mostly similar to the functions defined for L1 caches and memory.

As in the case of L1 caches, when an upgrade request is made, an association between the line address and the cache slot where the line will reside has to be maintained. Again, this can be done either by sending the slot number with the upgrade requests, or searching the requests from the lower levels which are waiting to be served (as done by *searchAddr* function). The number of additional bits needed is




(a) Modules in an Internal cache

| Buffer | FIFO | Description | Format |
|---|---|---|---|
| *rqFromC* | Yes | Requests from children | ⟨src-cache, address, old-state, upgrade-state⟩ |
| *rsFromC* | Yes | Responses from children | ⟨src-cache, address, downgrade-state, data⟩ |
| *toC* | Yes | Messages to children | ⟨dest-cache, ⟨**Req**, address, downgrade-state⟩⟩ \| ⟨dest-cache, ⟨**Rs**, address, upgrade-state, data⟩⟩ |
| *rqToP* | Yes | Requests to parent | ⟨parent-cache-id, current-cache-id, address, old-state, upgrade-state⟩ |
| *rsToP* | Yes | Responses to parent | ⟨parent-cache-id, current-cache-id, address, downgrade-state, data⟩ |
| *fromP* | Yes | Messages from parent | ⟨**Req**, address, downgrade-state⟩ \| ⟨**Rs**, address, upgrade-state, data⟩ |
| *cRqs* | No | Requests from children | ⟨src-cache, address, old-state, upgrade-state⟩ |
| *pRqs* | No | Requests from parent | ⟨address, downgrade-state⟩ |

(b) Buffers in internal cache, and its message formats

| Function | Description |
|---|---|
| $searchTags(cs, tags, a)$ | Similar to *searchTags* function for L1 caches. |
| $searchAddr(cRqs.all, a)$ | Similar to *searchAddr* function for L1 caches. |
| $searchSlot(cRqs.all, l)$ | Similar to *searchSlot* function for L1 caches. |
| $getVictim$ $(cRqs.all, pRqs.all, dirList, a)$ | Similar to the *getVictim* function of L1 caches. In addition to the conditions in L1 caches, the directory states of all the children for the victim, if any, must be $I$. |
| $completed$ $(cRqs.all, x, a)$ | Returns if there any "completed" requests in *cRqs* buffer, *i.e.,* a request $\langle c, a, y, x' \rangle$ in augmented state ⟨**WaitSt**, _⟩ for which $x \geq x'$. |
| $findHigher$ $(dirList, dirwList, x)$ | Returns a child cache $c$ whose directory state ($dirList[c]$) is higher than $x$ state, and where there is no pending response from that child ($i.e., \neg dirwList[c]$). |
| $isCompatible(x, y)$ | Similar to the function *isCompatible* for memory. |
| $findIncompatible$ $(c, dirList, dirwList)$ | Similar to the function *findIncompatible* for memory. |

(c) Helper functions in internal cache transitions

Figure 4-9: Internal Cache

**Module**:
**Regs** *cs*[NUM]({*I*}), *w*[NUM]({False}), *tags*[NUM]({_}), *data*[NUM]({_}), *dir*[NUM]({{*I*}}), *dirw*[NUM]({{False}});

**Rule** *CTransfer*:
```
let⟨c, a, y, x⟩ = rqFromC.pop;
cRqs.ins(⟨c, a, y, x⟩);
```

**Rule** *Hit*:
```
let n = cRqs.get(cs, tags, w, dir, dirw, pRqs.all);
let⟨Init, ⟨c, a, y, x⟩⟩ = cRqs.extract(n);
let l = searchTags(cs, tags, a);
when((l ≠ ϵ) ∧ (dir[l][c] ≤ y) ∧ (cs[l] ≥ x)
        ∧ (∀i ≠ c. compat(dir[l][i], x)) ∧ ¬dirw[l][c]
        ∧ (searchSlot(cRqs.all, l) = ϵ)
        ∧ (searchAddr(cRqs.all, a) = ϵ)
        ∧ (searchAddr(pRqs.all, a) = ϵ)) ⇒
    dir[l][c] := x;
    cRqs.upd(n, Free);
    toC.enq(⟨c, ⟨Rs, a, x,
            if (dir[l][c] = I) then data[l] else _⟩⟩);
```

**Rule** *MissByState*:
```
let n = cRqs.get(cs, tags, w, dir, dirw, pRqs.all);
let⟨Init, ⟨c, a, y, x⟩⟩ = cRqs.read(n);
let l = searchTags(cs, tags, a);
when((l ≠ ϵ) ∧ (dir[l][c] ≤ y) ∧ ¬((cs[l] ≥ x)
        ∧ (∀i ≠ c. compat(dir[l][i], x)))
        ∧ (searchSlot(cRqs.all, l) = ϵ)
        ∧ (searchAddr(cRqs.all, a) = ϵ)
        ∧ (searchAddr(pRqs.all, a) = ϵ)) ⇒
    cRqs.upd(n, ⟨WaitSt, l⟩);
```

**Rule** *MissByLine*:
```
let n = cRqs.get(cs, tags, w, dir, dirw, pRqs.all);
let⟨Init, ⟨c, a, y, x⟩⟩ = cRqs.read(n);
let l = getVictim(cRqs.all, pRqs.all, [], a);
when((l ≠ ϵ) ∧ (searchTags(cs, tags, a) = ϵ)
        ∧ (searchAddr(cRqs.all, a) = ϵ)
        ∧ (searchAddr(pRqs.all, a) = ϵ)) ⇒
    cRqs.upd(n, ⟨WaitV, l⟩);
```

**Rule** *DwnRqEvict*:
```
let n = cRqs.get(cs, tags, w, dir, dirw, pRqs.all);
let⟨⟨WaitV, l⟩, ⟨c, a, y, x⟩⟩ = cRqs.read(n);
let i = findHigher(dir[l], dirw[l], I);
when((i ≠ ϵ) ∧ (dir[l][i] > I)) ⇒
    toC.enq(⟨c, ⟨Req, a, I⟩⟩);
    dirw[l][i] := True;
```

**Rule** *DwnRs*:
```
let⟨c, a, x, v⟩ = rsFromC.pop;
let l = searchTags(cs, tags, a);
if (dir[l][c] = M) then data[a] := v;
dir[l][c] := x;
dirw[l][c] := False;
```

**Rule** *Writeback*:
```
let n = cRqs.get(cs, tags, w, dir, dirw, pRqs.all);
let⟨⟨WaitV, l⟩, ⟨c, a, y, x⟩⟩ = cRqs.read(n);
let a' = tags[l];
when(∀i. dir[l][i] = I) ⇒
    if (cs[l] ≠ I) rsToP.enq(⟨pid, ⟨cid, a', I,
                if (cs[l] = M) then data[l] else _⟩⟩);
    cs[l] := I;
    cRqs.upd(n, ⟨WaitSt, l⟩);
```

**Rule** *UpgRq*:
```
let n = cRqs.get(cs, tags, w, dir, dirw, pRqs.all);
let⟨⟨WaitSt, l⟩, ⟨c, a, y, x⟩⟩ = cRqs.read(n);
when(¬w[l] ∧ (cs[l] < x)) ⇒
    rqToP.enq(⟨pid, ⟨cid, a, cs[l], x⟩⟩);
    w[l] := True;
```

**Rule** *UpgRs*:
```
let⟨Rs, a, x, v⟩ = fromP.pop;
let l = searchAddr(cRqs.all, a);
when(searchAddr(pRqs.all, a) = ϵ) ⇒
    if (cs[l] = I) {data[l] := v; tags[l] := a;}
    cs[l] := x;
    w[l] := False;
```

**Rule** *DwnRq*:
```
let n = cRqs.get(cs, tags, w, dir, dirw, pRqs.all);
let⟨⟨WaitSt, l⟩, ⟨c, a, y, x⟩⟩ = cRqs.read(n);
let i = findIncompatible(c, dir[l], dirw[l], x);
when(i ≠ ϵ) ⇒
    toC.enq(⟨i, ⟨Req, a, if (x = M) then I else S⟩⟩);
    dirw[l][i] := True;
```

**Rule** *Deferred*:
```
let n = cRqs.get(cs, tags, w, dir, dirw, pRqs.all);
let⟨⟨WaitSt, l⟩, ⟨c, a, y, x⟩⟩ = cRqs.extract(n);
when((cs[l] ≥ x) ∧ ¬dirw[l][c]
        ∧ (∀i ≠ c. compat(dir[l][i], x))) ⇒
    dir[l][c] := x;
    cRqs.upd(n, Free);
    toC.enq(⟨c, ⟨Rs, a, x,
        if (dir[l][c] = I) then data[l] else _⟩⟩);
```

**Rule** *PTransfer*:
```
let⟨Req, a, x⟩ = fromP.pop;
pRqs.ins(⟨a, x⟩);
```

**Rule** *Drop*:
```
let n = pRqs.get(cs, tags, w, dir, dirw, cRqs.all);
let⟨a, x⟩ = pRqs.read(n);
let l = searchTags(cs, tags, a);
when((l = ϵ) ∨ (cs[l] ≤ x)) ⇒
    pRqs.remove(n);
```

**Rule** *PProcess*:
```
let n = pRqs.get(cs, tags, w, dir, dirw, cRqs.all);
let⟨a, x⟩ = pRqs.read(n);
let l = searchTags(cs, tags, a);
when((cs[l] > x) ∧ (∀i. dir[l][i] ≤ x)
        ∧ ¬completed(cRqs.all, cs[l], a)) ⇒
    pRqs.remove(n);
    rsToP.enq(⟨pid, ⟨cid, a, x,
            if (cs[l] = M) then data[l] else _⟩⟩);
    cs[l] := x;
```

**Rule** *DwnRqP*:
```
let n = pRqs.get(cs, tags, w, dir, dirw, cRqs.all);
let⟨a, x⟩ = pRqs.read(n);
let l = searchTags(cs, tags, a);
let i = findHigher(dir[l], dirw[l], x);
when((i ≠ ϵ) ∧ ¬completed(cRqs.all, cs[l], a)) ⇒
    toC.enq(⟨i, ⟨Req, a, x⟩⟩);
    dirw[l][i] := True;
```

Figure 4-10: Module $L_{int}$': Module containing the transition rules of the $L_{int}$ cache

again lg(cache-associativity).

The FIFO buffers in Figure 4-9b are similar to the FIFO module in Figure 2-1. The non-FIFO buffer *cRqs* is similar to the *cRqs* buffer of the L1 cache, and each entry goes through the same set of stages as shown in Figure 4-5. The non-FIFO buffer *pRqs* is also similar to its namesake in the L1 caches.

### 4.2.6 Subtle Design Decisions in the Caches

In this section, we will highlight some of the subtleties in the design of the caches. In the verification of the cache-coherence protocol (which we discuss in Chapter 8), these subtleties lead to invariants that are proven.

The first thing to note is that the cache state changes only on sending responses to the parent or receiving responses from the parent. Similarly, the directory state changes only on sending responses to the corresponding child or receiving responses from the corresponding child.

For a given address, there can be only one request from a child (or processor) in the *cRqs* buffer whose entry is in a waiting augmented state of ⟨WaitSt, _⟩ or ⟨WaitV, _⟩. This is guaranteed by the guards of the rules that change the augmented state of a request entry to ⟨WaitSt, _⟩ or ⟨WaitV, _⟩ (using function *searchAddr*(*cRqs.all*, ...)). In case the cache has to evict a slot to accommodate a new address, the address of the slot it evicts is also not going to be present in the *cRqs* buffer in a request whose entry is in a waiting augmented state of ⟨WaitSt, _⟩ or ⟨WaitV, _⟩.

A request entry in the *cRqs* buffer can go into a waiting augmented state of ⟨WaitSt, _⟩ or ⟨WaitV, _⟩ from Init or Free only when there are no requests for the same address in the *pRqs* buffer. This is again guaranteed by the guards of the respective rules (using function *searchAddr*(*pRqs.all*, ...)).

Since the cache can evict an address (using **Rule**⟨*Writeback*⟩) without its parent actually asking the cache to downgrade, the parent will not be informed of the downgrade till it receives this unsolicited response sent by the cache. So, the parent could have sent a downgrade request in the interim, which must be discarded. **Rule**⟨*Drop*⟩

does just that.

Whenever a cache sends a request to its parent, it sets its wait state. Similarly, whenever it sends a request to its child, it sets the corresponding child's directory-wait state. When a cache is in wait-state, handling a downgrade request from its parent is not straightforward. We have to decide whether to handle the request or ignore it for the time being. Similarly, on the parent's side, we need to decide whether to handle or ignore an upgrade request from a child, when the corresponding directory-wait state is set.

Let's say we decide to handle neither. This will clearly create a deadlock. Let's say we decide to handle both. This can lead to a livelock as follows: The parent sends an upgrade response, and the child sends a downgrade response. But, the parent immediately sends a downgrade request again (it must have sent a downgrade request earlier for a reason, which still remains true), and the child immediately sends an upgrade request again. Once again, the respective requests can be processed without handling the cause that made the two entities send the respective requests, leading to a livelock.

To avoid the problems of livelock and deadlock, we have to allow only one of them to handle the request while the other should ignore the request. The downgrade request from a parent is always handled except when there is a pending request in the *cRqs* buffer that can be "completed" (as given by function $completed(cRqs.all, \ldots)$). In case of the latter, the request from the parent is ignored. This is acceptable because the cache can make forward progress by servicing the pending request in the *cRqs* buffer that can be completed. We will prove in Chapter 8 that this avoids livelocks. In fact, it is easy to see that handling the parent's request when a pending request in the *cRqs* buffer can be completed can lead to a livelock.

Let's say two caches have pending requests to upgrade to $M$ in the *cRqs* buffer for the same address, and let's say the cache state of the first is $M$ and the second is $I$. So the second cache sends an upgrade-to-$M$ request, which is received by the parent. The parent then sends a downgrade-to-$I$ request to the first cache. The first cache downgrades its cache state to $I$, sending a downgrade response to the parent.

The parent, on receiving the response, sends an upgrade-to-$M$ response to the second cache. The second cache upgrades its cache state to $M$ on receiving the response. This scenario is symmetric to the one we started out with. Doing the symmetric set of actions will lead to the exact scenario we started out with, creating a livelock.

We will be showing all the invariants that are held by the caches formally in Chapter 8, and show how they help in verifying that the hierarchy of caches implements the atomic memory.

## 4.3 Network Between the Caches

In this section we will give the constraints on the implementation of the network connecting the cache nodes.

For messages from parent to children, each cache's *toC* buffer is logically connected to each of its children's *fromP* FIFO buffer. Similarly, for requests from children to their parents, each child's *rqToP* buffer is logically connected to the corresponding parent's *rqFromC* buffer, and for responses from children to their parents, each child's *rsToP* buffer is logically connected to the corresponding parent's *rsFromC* buffer.

The network can be implemented using any physical topology as long as the following three different types of messages, *viz.,* (a) messages from parent to children, (b) requests from children to parent and (c) responses from children to parent do not interfere with each other. Moreover, even within the same type of messages, the messages transmitted between a pair of levels of the cache hierarchy should not interfere with the messages transmitted between a different pair of levels, except for the responses from children to parent, which can interfere across hierarchies. By not interfering, we mean that, if one of these messages is blocked from sending because the buffers carrying those types of messages is full, it shouldn't affect the ability to send other kinds of messages.

Another requirement that the network carrying these different types of message should obey is that there is point-to-point ordering for messages sent between the same source-destination pair of buffers (even though messages having different source-

destination pair of buffers even among the same pair of caches have no ordering requirement between them.)

A simple implementation of the network to satisfy the above requirements is to connect each of the buffers directly to the appropriate buffers in the parent or the children. Such a connection would involve a transition rule for each pair of connected buffers, as shown below.

**Rule** *Connect1*:

$a = x.pop;$
$y.enq(a);$

However, as the number of cores increase, having such a dedicated point to point network between all the communicating buffers in the caches is prohibitively expensive. Most modern processors implement a separate physical interconnect network, organized as a ring, mesh, *etc.*, which connects every cache and every processor. The logical tree topology can be implemented on top of this physical network. In in order to implement the constraints on the interference of messages on a single physical network, one has to use the concept of *virtual channels* or *virtual networks* [81]. Each virtual channel is implemented using buffers that store only messages flowing in that channel, and is therefore logically independent of other virtual channels.

Section 4.2.4 mentions two alternative implementations for requests and responses going from children to parents: (a) using two separate buffers, one for requests and the other for responses; and (b) using the same buffer with high priority for responses and low priority for requests. While we have implemented the protocol assuming the former, if one chooses to use the latter, then it is the duty of the network to guarantee that high priority messages are never blocked by low priority messages. This can also be implemented using virtual channels, but every buffer (including those in the switches, *etc.*) in the virtual channel must obey this priority.

## 4.4 Design for Provability

The designs of the speculative out-of-order processor and the cache hierarchy involve several abstractions. For instance, in the case of the processor, the reorder buffer and branch predictor units were abstracted as modules with constraints on their behavior (in the case of the branch predictor module, there were no constraints on its behavior). Similarly, in the case of the cache hierarchy, we specified functions such as *getVictim*, *searchTags*, *etc.* by giving a specification of the values returned by the function instead of actually defining the function itself.

Let us delve into the reorder buffer module of the processor. This module defines several methods (as shown in Section 3.3.1). While we do not give the definitions of any of these methods, we do constrain the behavior of the *oldest* method alone using Invariant 1. The rest of the methods are allowed to be implemented in any manner, and it does not affect the correctness of the reorder buffer module. Moreover, the reorder buffer module can have any number of internal rules, as long as Invariant 1 is obeyed.

Let us now look into Invariant 1. It constrains the values returned by the *oldest* method of the reorder buffer. In order to specify the constraint, we have used the same *dec* and *exec* functions which we had used in the specification of the decoupled processor, $P_{\text{dec}}$. This brings up another point of abstraction that we employed in this processor design: reuse of the function both in the implementation module and the specification module. In fact, **Rule**⟨*LoadRsBad*⟩ of the speculative out-of-order processor, $P_{\text{so}}$, also uses the same *exec* function. Use of the same function in both the implementation and the specification avoids the need to verify the details of the decode and execution units. For instance, we need not verify that a "plus" operation is implemented correctly in the out-of-order processor but we can directly conclude that $P_{\text{so}}$ behaves exactly like $P_{\text{dec}}$ for any instantiation of the *dec* and *exec* functions, irrespective of whether the "plus" operation has been actually implemented correctly.

Thus, if one wants to verify that a particular ISA has been implemented correctly, then one has to make sure that the *dec* and *exec* functions are consistent with the instructions of the ISA like "plus".

Compared to the reorder buffer module, the branch predictor module does not have to obey any specification – it can return any value as the next predicted *pc*. This is acceptable since the reorder buffer processor has a mechanism to correct itself in case of a bad prediction.

In summary, in our design of the speculative out-of-order processor, we employed two kinds of abstractions:

1. Abstracting using a logical specification: A concrete function or module is replaced with a logical specification of its behavior.

2. Using identical functions in implementation and specification.

As mentioned earlier, the cache hierarchy also employs several abstractions, which manifest themselves in the form of functions defined in Figure 4-3c, Figure 4-6c and Figure 4-9c. All these functions were specified in terms of the constraints on the values returned by these functions instead of giving an actual definition. This allows us to change several implementation parameters (like cache associativity, replacement policy, *etc.*) without having to change any of the proofs – the proofs are all orthogonal to such implementation parameters.

In addition to keeping the proofs more general than restricting them to specific instantiations of the implementations, the use of abstractions has another important consequence. The use of abstractions is akin to specifying a general mechanism that supports any specific policy, and the policies are left unspecified. This reduces the proof burden considerably, as the design is not cluttered with unnecessary details orthogonal to the proof. One can imagine the cluttering in the code corresponding to a cache design which has a hard-coded replacement policy for a specific cache-

organization. The definitions of functions such as *getVictim* and *searchTags* will be inlined at their call sites, and if the functions get complicated (for instance, *getVictim* can get very complicated depending on the sophistication of the replacement policy), then the main gist of the transitions that happen in the rules (which were not more than 10 lines in each of Figures 4-4, 4-7 and 4-10) will be lost in the details of the functions' implementations.

In summary, the use of abstractions has two purposes when it comes to verification:

- to parameterize the proofs over several concrete instantiations of the implementations, and
- to reduce the verification burden by leaving the policies unspecified and instead verifying the design for any policies.

The second point is very important especially for verification flows involving theorem proving, such as ours. This is because one needs to manually supply the proofs for the designs, which is easier when unnecessary details are absent from the design.

While good software engineering discipline dictates that such abstractions in designs are useful, it is uncommon to find them in practice, for hardware designs. For instance, the function *getVictim* is typically inlined in actual hardware design. In fact, coming up with the right abstractions during the design phase can be hard. But, as we argue in this section, spending effort to employ right abstractions in the design pays off during its verification. Otherwise, during the verification phase, one has to first extract out the policies that are irrelevant from a correctness perspective before attempting to verify the system.

This notion of "design for provability" or "design for verifiability" is not new. While some work on the methodology of design for verifiability [18, 19] restricts the designs to use only a subset of the system description language like VOVHDL [17], our methodology is more informal and along the lines of Curzon *et al.* [25] and Milne *et*

*al.* [64].

## 4.5 Conclusion

In this chapter we presented a specification of a cache coherence protocol in terms of transition rules of cache nodes. This hierarchy of caches executing this protocol along with the interconnect network forms the memory subsystem of any modern processor. It remains to be shown that this system indeed implements the atomic memory specification $M_{\text{atomic}}$. We will embark on this task in the next part of the dissertation, starting with the formalization of the semantics of Bluespec modules.

# Part II

# Verifying Hardware Systems using Labeled Transition Systems

# Organization

In this part of the dissertation we will formalize the notion of *refinement* and state several properties of the refinement relation. We will later use these properties to prove that the multiprocessor system consisting of speculative out-of-order cores connected to a coherent cache hierarchy implements sequential consistency.

We will start by giving an overview of the formal verification techniques in the hardware domain in Chapter 5. We will also compare and contrast verification using model-checking with that using proof assistants or theorem provers. Most of the formal proofs in this thesis were mechanically verified using the Coq proof assistant. We will discuss the advantages of using theorem provers for hardware verification.

We will start by giving the modular semantics of hardware modules written in Bluespec using *Labeled Transition Systems* (LTS) [65]. While the concept of LTSes is widely known, we show how a module in Bluespec can be interpreted as an LTS. The label in an LTS transition is used to represent communication that happens in Bluespec modules via methods. In Chapter 6, we specify the formal syntax for writing and composing hardware modules in Bluespec which we have been using informally in Part I. We will then specify its semantics in terms of LTSes. We will then specify the semantics of composition of Bluespec modules. We will also formally define the notion of *refinement* or *implementation*. We will state useful properties of refinements including composition properties and prove them.

The refinement methodology used in this thesis also bears similarity to those em-

ployed by earlier classical works [1,43,58]. These works were the first to introduce the notion of refinement relation between an implementation and its specification based on *abstraction functions.* They build the theory about refinements using *forward simulation* and *backward simulation* relations. This thesis uses the same notion of forward simulation. In addition, this thesis also formalizes a type of compositional reasoning and proofs using LTSes.

Next, we will give the formal proofs for the multiprocessor system implementing sequential consistency. In Chapter 7, we will give an overview of the overall proof. We will state and prove the relationships between the speculative out-of-order processor $P_{\text{so}}$, the decoupled processor $P_{\text{dec}}$ and the instantaneous processor $P_{\text{inst}}$.

In Chapter 8, we will give the formal proof that the hierarchy of caches $M_{\text{cache}}$ implements atomic memory $M_{\text{atomic}}$. We will also prove that the cache coherence protocol ensures that the cache hierarchy is free of deadlocks and livelocks.

Finally, in Chapter 9 we will present the work to be done in terms of improving the verification framework and building a library of formally verified hardware components using this technique.

# Chapter 5

# Background: Verification Techniques for Hardware Systems

We will begin this chapter with an overview of the model-checking based hardware verification techniques which are widely in use, along with some of their limitations, in Section 5.1. In Section 5.2, we will give some of the instances of hardware verification carried out using theorem provers or proof assistants and discuss their trade-offs compared to model-checking.

## 5.1 Hardware Verification using Model-Checking

Designers recognize the challenge of implementing hardware systems, to such an extent that they have already become some of the most serious real-world adopters of formal methods. Hardware verification is dominated by model checking; for instance, processor verification [15, 61] and more recently, Intel's execution cluster verification [51].

Typical hardware systems are parameterized over numerous design parameters. In our multiprocessor system, the cache hierarchy is parameterized over the number of levels in the tree, number of children in each node, *etc.* In fact, it is parameterized

completely over the shape of the tree itself. More low-level components, like buffers, can also be parameterized over their sizes.

While many abstraction techniques are used to reduce parameterized designs to finite state spaces, which can be explored exhaustively, there are limits to the construction of sound abstractions. So verifications of, *e.g.*,, cache-coherence protocols have mostly treated systems with concrete topologies, involving particular finite numbers of caches and processors. For instance, explicit-state model checking tools like Murphi [36] or TLC [50, 57] are only able to handle single-level cache hierarchies with fewer than ten addresses and ten CPUs, as opposed to the billions of addresses in a real system, or the ever-growing number of CPUs. Symbolic model checking by itself does not do any better: McMillan *et al.* have verified a 2-level MSI protocol in the Gigamax distributed multiprocessor having two clusters with six processors each using SMV [63].

Optimizations on symbolic model checking (*e.g.,* partial order reduction [10], symmetry reduction [11, 21, 24, 37, 47, 90], compositional reasoning [48, 60, 62], extended-FSM [35], *etc.*) scale the approach, supporting some form of structural parameterization. For example, some of these techniques can verify up to two levels of cache hierarchy (*i.e.,* a memory at the root, and a set of L1 caches) with arbitrary number of processors, but they are still unable to handle realistic multi-level hierarchical protocols. An exception to this is work by Zhang *et al.* to verify hierarchical cache-coherence protocols [91]. This work, however, has two huge restrictions: (a) the cache hierarchy should exhibit a fractal-like behavior, *i.e.,* the composition of several caches should also behave like a single cache, and (b) the cache hierarchy can only be a binary tree. Even with these restrictions, they rely on paper-and-pencil proofs for theorems about compositions. The authors agree in a later work [90] that, as a result of the binary restriction, the protocol suffers from a serious performance handicap. It is imperative that no changes are made to the design to make it more amenable to

verification, let alone those that in fact reduce the performance! They also advocate that parameterization should be restricted to single dimensions (*e.g.*,, only the number of children of each cache can be parameterized, not the number of levels) for the state-of-the-art tools based on model checking to scale.

Even if the above mentioned optimization techniques are used in model-checking-based approaches, verification of realistic complex systems is still not a push of a button. Finding state invariants automatically is really hard for complex systems, and more often than not, these have to be supplied manually as part of the specification to verify against. For instance, Chou *et al.* [24] propose a counter-example guided abstraction refinement technique, where extra relaxed invariants, called "non-interference lemmas" are added manually, and whenever a model-checker returns a counter example for the manually supplied invariant, it is manually strengthened. In this way, all the invariants of the system are discovered, eventually verifying the system. Flow-based methodology [83] gives yet another way of manually specifying invariants for verifying cache-coherence protocols. It is based on supplying a single "flow" of messages that has to take place to complete a complex transaction.

Finally, model-checking provides no intuition as to why a protocol is correct, if the model checker successfully proves a protocol. Developing a new protocol, therefore, cannot use any intuition that could have been gained from the first protocol, requiring the verification to be carried out from scratch. However, this issue is somewhat mitigated using interactive model-checking based techniques, which, while not push-button, enable verification of more complex systems and help gather intuition about the design.

## 5.2 Hardware Verification using Theorem Provers

We propose a mechanized theorem-prover based approach to verifying systems. This solves the scalability problems faced by model checkers, and readily support arbitrary parameterization. People have used theorem provers to verify microprocessors before, *e.g.,* HOL to verify an academic microprocessor AVI-1 [89]. Cache-coherence proofs have also used mechanized theorem provers, though all previous work has verified only single-level hierarchies. Examples include using ACL2 for verifying a bus-based snoop protocol [66], using a combination of model-checking and PVS [71] to verify the FLASH protocol [55], and using PVS to mechanize some portions of a paper-and-pencil proof verifying that the Cachet cache-coherence protocol [80] does not violate the CRF memory model.

The main advantage of using theorem provers is that it enables us to verify a system even in the presence of parameters, instead of verifying just concrete instances. For example, in this thesis, we proved that the cache coherence protocol is correct over any arbitrary tree hierarchy.

But the downside of using theorem provers is that considerable effort is needed by verification experts to provide a proof for the system. The human verification effort increases with the complexity of the protocol – the verification engineer has to codify something akin to the paper-and-pencil proof inside the theorem prover, which the theorem prover will check and "certify". This human effort is comparable to that needed to supply manual invariants in the case of model-checking based techniques (like that of Chou *et al.* [24] and the flow-based methodology [83]). In this thesis, we try to mitigate some of these issues using two approaches:

1. By developing a systematic methodology for designing and verifying hardware systems using Bluespec specifications and the theory of Labeled Transition Systems.

2. By using the Coq proof assistant which has several features that make formal verification of hardware systems using theorem provers more practical.

The Coq proof assistant has intrinsic support for higher-order logic, which enables us to state and prove complex properties about systems as opposed to being restricted to Linear Temporal Logic (LTL), Computation Tree Logic (CTL) or CTL*. Such a logic system allows us to structure the proof in the same way as would be written by hand. In addition to the structural modularization of proofs, Coq also permits "programming the proof procedure" via what is known as *tactics* using the *LTac* language [34], and a vast library of tactics which ease the proofs. For instance, if two completely unrelated theorems are both proven using induction and the assumptions about induction hypotheses, a tactic `induction; simpl; assumption` would prove both of these unrelated theorems. Finally, the LTac language also permits "proof searches by pattern recognition" and supplying tactics for the matched patterns. This enables us to search for a particular pattern and discharge the proof obligations for such patterns using the tactics supplied for that pattern. This is especially useful for discharging trivial proof obligations and concentrating on the meat of the proof.

While this thesis exclusively uses the Coq proof assistant for mechanized verification, it is conceivable that other theorem provers also offer the same advantage as Coq or perhaps are even more suitable for hardware verification. Adam Chlipala [23] compares Coq (favorably) against other theorem provers that are currently in use.

## 5.3 Conclusion

In this chapter we gave an overview of the hardware verification landscape, where most formal verification is dominated by model-checking. We also argued why we chose to use theorem prover based approach for hardware verification and listed some of the reasons for using the Coq proof assistant. In the next chapter we will give the

formal modular semantics of Bluespec using Labeled Transition Systems and in the later chapters, verify our multiprocessor system example.

# Chapter 6

# Modular Semantics of Bluespec

In this chapter we will formalize the syntax and semantics of Bluespec. First, we will specify the syntax for Bluespec modules in Section 6.1. In Section 6.2, we will discuss the behavior of a composition of modules by constructing a single *primitive module* from the composition. This will serve as the guideline to define the semantics of modules. In Section 6.3, we discuss the *modular semantics* of modules. We start by giving semantics for actions, which are the building blocks for rules and methods of a module. We then present the semantics of a primitive module, followed by the semantics for a composition of modules. The semantics for a composition of modules are similar to those of Labeled Transition Systems (LTSes) [65]. We will then give the formal definition of the "implements" or "refines" relation in 6.4. Finally, in Section 6.5, we will state some of the properties of the semantics and the implements relation.

## 6.1 Syntax

We will start with expressions in this language which correspond to register reads, constant reads or operations on other expressions. For notational convenience, we use the symbol $\vec{x}$ to represent a list of elements $x$. Henceforth we will use $r$ and $c$ as representative symbols for register names and constants, respectively, and $\mathcal{R}$ and $\mathcal{C}$

to represent the set of register names and set of constants, respectively. We will use $f, g$ or $h$ to represent method names and $\mathcal{F}$ to represent the set of method names. We use $k$ for rule names and $\mathcal{K}$ to represent the set of rule names. We use $\mathcal{E}, \mathcal{A}$ and $\mathcal{M}$ to represent the sets of expressions, actions and modules, respectively.

$$
\begin{array}{lcl}
\text{Expression } \mathcal{E} & ::= & r \\
& | & x \\
& | & c \\
& | & \textbf{op } \vec{e}
\end{array}
$$

An action is formally defined below using the Continuation Passing Style (CPS) syntax [74]. An action can be a call of a method of a different module passing an *expression* as an argument, a variable-to-expression binding, a register update, a conditional action, an assertion or a guarded action, or a return expression. Each action except the return action contains a continuation action. Note that a method call is an action returning a value (called an *action value* in Bluespec) and its effect, informally, is to update some state in another module in addition to returning a value. There is an explicit check to ensure that each method is called only and each register is written only once within an action.

$$
\begin{array}{lcl}
\text{Action } \mathcal{A} & ::= & r := e;\ a \\
& | & \textbf{let}\, x = f(e);\ a \\
& | & \textbf{let}\, x = e;\ a \\
& | & \textbf{if}\ (e)\ \textbf{then}\ a\ \textbf{else}\ a;\ a \\
& | & \textbf{when}(e) \Rightarrow\ a \\
& | & \textbf{return}(e)
\end{array}
$$

A module is either a *primitive module* comprising of registers along with their initial values, rules and methods; or it is a composition of several primitive modules. Rules

are named actions, and methods are named function abstractions for actions.

$$\begin{array}{lcl}\text{Module } \mathcal{M} & ::= & \langle \overrightarrow{\langle r, c \rangle}, \overrightarrow{\langle k, a \rangle}, \overrightarrow{\langle f, \lambda x.a \rangle} \rangle \\ & | & m + m\end{array}$$

We ensure that no two registers, two rules or two methods defined in a module have the same name. This means that every register, rule and method name in every primitive module present in a composition is globally unique. We also ensure that registers being accessed in a rule of a primitive module are defined in that module and the methods being accessed in any action of a primitive module are *not* defined in that module. We ensure that there is no cycle in the call graph of a module (*i.e.*, a graph in which the vertices are the set of methods defined by a module and each edge between two methods represents the call of one method inside another). Finally, we ensure that only one primitive module can call a method defined in another primitive module. It is easy to check for these conditions syntactically, and we will assume that this check has been done for the rest of this dissertation. As mentioned early on in this dissertation (in Section 2.1), we omit specification of types; though in the real implementation in Coq, appropriate type checking is done.

The return values of the actions corresponding to rules and corresponding to methods which do not return any values are ignored. For syntactic convenience, we do not write the return statements for rules or such methods.

We can define several functions that obtain various components from the definition of a module. These will be used later in this chapter.

Function **getRegs** gets the set of registers along with their initial values from a composition of modules.

**Definition 10.**

$$\mathsf{getRegs}\ m \triangleq \begin{cases} regs & : \quad m = \langle regs, rules, meths\rangle \\ (regs\ m_1) \cup (regs\ m_2) & : \quad m = m_1 + m_2 \end{cases}$$

Function **getRules** gets the set of rules in a composition of modules.

**Definition 11.**

$$\mathsf{getRules}\ m \triangleq \begin{cases} rules & : \quad m = \langle regs, rules, meths\rangle \\ (\mathsf{getRules}\ m_1) \cup (\mathsf{getRules}\ m_2) & : \quad m = m_1 + m_2 \end{cases}$$

Function **getMethods** gets the set of methods along with their bodies from a composition of modules.

**Definition 12.**

$$\mathsf{getMethods}\ m \triangleq \begin{cases} meths & : \quad m = \langle regs, rules, meths\rangle \\ (\mathsf{getMethods}\ m_1) \cup (\mathsf{getMethods}\ m_2) & : \quad m = m_1 + m_2 \end{cases}$$

Function **domain** gets the domain of values in a finite map (represented as a key-value pair).

**Definition 13.**

$$\mathsf{domain}\ xs \triangleq \{k | \langle k, v\rangle \in xs\}$$

Function **getDefs** gets the set of method names defined in a composition of modules using the **domain** function.

**Definition 14.**

$$\mathsf{getDefs}\ m \triangleq \mathsf{domain}\ (\mathsf{getMethods}\ m)$$

For obtaining the called methods in a module, we need to get a list of all the methods called by every rule or method in a module. Function $\mathsf{getCalls}_a$ obtains the

list of methods called in an action, from which we can obtain **getCalls**, the list of methods called in a module.

**Definition 15.**

$$\mathsf{getCalls}_a\ a \quad = \quad \begin{cases} \mathsf{getCalls}_a\ a_1 : a = r := e; a_1 \\ \mathsf{getCalls}_a\ a_1 : a = \mathbf{when}(e) \Rightarrow a_1 \\ \{\} : a = \mathbf{return}(e) \\ \mathsf{getCalls}_a\ a_1 : a = \mathbf{let}\, x = e; a_1 \\ \mathsf{getCalls}_a\ a_1 \cup \mathsf{getCalls}_a\ a_2 \cup \mathsf{getCalls}_a\ a' : a = \mathbf{let}\, x = \mathbf{if}\ (e)\ \mathbf{then}\ a_1\ \mathbf{else}\ a_2; a' \\ \{f\} \cup \mathsf{getCalls}_a\ a_1 : a = \mathbf{let}\, x = f(e); a_1 \end{cases}$$

**Definition 16.**

$$\mathsf{getCalls}\ m \quad = \quad \begin{cases} \{g | \langle R, a\rangle \in \mathit{rules}, g \in \mathsf{getCalls}_a\ a\} \cup \{g | \langle f, \lambda x.a\rangle \in \mathit{meths}, g \in \mathsf{getCalls}_a\ a\} \\ \qquad : \quad m = \langle \mathit{regs}, \mathit{rules}, \mathit{meths}\rangle \\ (\mathsf{getCalls}\ m_1) \cup (\mathsf{getCalls}\ m_2) \quad : \quad m = m_1 + m_2 \end{cases}$$

## 6.2 Meaning of Module Composition

We will now describe the behavior of a composition of modules using the concept of inlining. It can essentially be viewed as a syntactic operation which inlines definitions of the called methods at the places of call. Once a method has been called by some module in the composition of modules, it can no longer be called by any other module; the method is effectively *hidden*.

We will now give the general procedure for inlining modules. We use $[\![m]\!]$ to denote

the inlining function applied on a module. We read $[\![m]\!]$ as the *inlined version of m*.

$$\begin{aligned}[\![m]\!] \;\triangleq\; & \langle \mathit{regs}, \mathit{rules}[\mathit{meths}], \mathit{meths}[\mathit{meths}] / (\mathsf{getCalls}\ m)\rangle \\ \textbf{where} \quad \mathit{regs} &= \mathsf{getRegs}\ m \\ \mathit{rules} &= \mathsf{getRules}\ m \\ \mathit{meths} &= \mathsf{getMethods}\ m\end{aligned}$$

The operator $\mathit{actions}[\mathit{meths}]$ inlines the body of any method $f$ called in *actions* which is present in *meths*. We overload the inlining operator to work on both rules and method bodies. The meaning of the $/$ operator is defined below, where $\pi_i$ refers to the $i^{\text{th}}$ element in a tuple.

$$xs / ys \triangleq \{x | x \in xs, \pi_1\ x \notin ys\}$$

Using the $/$ operator, we are effectively hiding all the methods which are both defined and called within the same module $m$.

We will now show an example of this inlining procedure by composing two primitive modules. The first module is the instantaneous processor module, $P_{\text{inst}}$.

We will connect this processor to the first port of a 2-ported version of memory $M_{\text{inst}}$ (defined in Figure 3-2). Looking at the definition of $M_{\text{inst}}$ in Figure 3-2, one can see that the 2-ported version of the memory has four methods: $ldRq^1$ and $stRq^1$ corresponding to port-1, and $ldRq^2$ and $stRq^2$ corresponding to port-2.

We will now show the primitive module $[\![P^1_{\text{inst}} + M_{\text{inst}}]\!]$ in Figure 6-1.

As can be seen in Figure 6-1, the resulting primitive module has the method definitions (of $M_{\text{inst}}$) inlined in the places where they are called (in the rules of $P^1_{\text{inst}}$). The methods of $M_{\text{inst}}$ that are not called anywhere ($ldRq^2$ and $stRq^2$) are exposed as methods of the resulting primitive module.

```
Module:
Regs pc(pc_0), s(s_0), mem[NUM](mem_0);

Rule NonMemory^1:
let i = getInst(pc);
let⟨Nm, x⟩ = dec(s, pc, i);
let⟨s', pc'⟩ = exec(s, pc, ⟨Nm, x⟩);
s := s';
pc := pc';

Rule Halt^1:
let i = getInst(pc);
let⟨Halt⟩ = dec(s, pc, i);
halt^1;

Rule Load^1:
let i = getInst(pc);
let⟨Ld, va, r_v⟩ = dec(s, pc, i);
let v = mem[va];
let⟨s', pc'⟩ = exec(s, pc, ⟨Ld, r_v, v⟩);
s := s';
pc := pc';

Rule Store^1:
let i = getInst(pc);
let⟨St, va, v⟩ = dec(s, pc, i);
mem[va] := v;
let⟨s', pc'⟩ = exec(s, pc, ⟨St⟩);
s := s';
pc := pc';

Meth ldRq^2(va):
return(mem[va]);

Meth stRq^2(va, v):
mem[va] := v;
```

Figure 6-1: Primitive Module given by $[\![P^1_{\text{inst}} + M_{\text{inst}}]\!]$

## 6.3 Modular Semantics

In this section, we will give the modular semantics of expressions, actions and modules. We start with the semantics for expressions and actions.

The goal of giving formal semantics for Bluespec programs is to understand the behavior of Bluespec programs and to give a procedure to compare two programs; in particular to check if one program refines another. The semantics should be such that it should capture the intuitive notion of refinement between two programs. Modular semantics helps take the notion of refinement one step further: whenever one module refines another module, then the latter can replace the former in any context. This allows us to break the proofs about the refinement of a big system into its component modules, just the way a designer would build the big system in the first place.

### 6.3.1 Semantics for Expressions

We first give the deterministic, denotational semantics of evaluating an expression when given a map of register values. The notation $[\![\mathbf{op}]\!]$ refers to a function which is semantically equivalent to the operation performed by **op** on the list of expressions, taking the current finite register mapping $o$ as an argument. The operation $o(r)$ returns the value of register $r$ from the mapping $o$ and is defined in Section A.1.

**Definition 17.**

$$\begin{aligned}
[\![r]\!]\ o &= o(r) \\
[\![x]\!]\ o &= _\ \text{(this cannot occur)} \\
[\![c]\!]\ o &= c \\
[\![\mathbf{op}(es)]\!]\ o &= [\![\mathbf{op}]\!]\ (\mathsf{map}\ (\lambda e.([\![e]\!]\ o))\ es)
\end{aligned}$$

We will never access a register which is not present in the finite map $o$ since every expression and action inside a primitive module only accesses the registers defined within the primitive module. We will also never encounter a variable when we are

evaluating an expression; whenever the semantics of an action is derived, it substitutes the values for every variable in the expressions present in the action before evaluating the expression.

### 6.3.2 Semantics for Actions

Next we give the semantics of an action $a$. The judgment for an action collects the state updates performed by the action, the methods called by the action along with their arguments and return values and computes the value returned by the action. Unlike the semantics of expressions, which computes the value of an expression once the register map is supplied, actions cannot compute any value with just a register map. This is because the return values associated with the called methods cannot be computed from the action itself; they will be supplied externally by a module which defines the corresponding method. Thus we present the semantics of an action in the form of a relation; the return values of a method call are unconstrained free variables in the relation. The judgments are of the form $\left(o \vdash (a) \Downarrow \langle u, cs, v\rangle\right)$ where $o$ is a finite map of register names to values giving the values of the registers before the action has performed any updates, $u$ is the set of updates that the action performs on the registers, $cs$ is the set of methods called by the action (along with their arguments and return values) and $v$ is the value computed by the action. We read $\left(o \vdash (a) \Downarrow \langle u, cs, v\rangle\right)$ as an *action $a$ executing on state $o$, performing updates $u$, calling methods $cs$ and returning value $v$.*

We use $a \# b$ to denote that the two sets $a$ and $b$ are disjoint. The sets can also be made of tuples whose first elements are keys, in which case $a \# b$ denotes that the keys are disjoint. Formally, $a \# b \triangleq (\mathsf{map}\ \pi_1\ a \cap \mathsf{map}\ \pi_1\ b) = \{\}$. We need to check disjointness conditions to avoid calling the same method twice, or updating the same register twice, as can been seen in the following judgments.

**Definition 18** (Action).

$$\text{ActionWriteReg}\ \frac{\left(o \vdash (a) \Downarrow \langle u, cs, v\rangle\right) \qquad \langle r, _\rangle \notin u}{\left(o \vdash (r := e;\ a) \Downarrow \langle \langle r, [\![e]\!]\ o\rangle :: u, cs, v\rangle\right)}$$

$$\text{ActionCall}\ \frac{\left(o \vdash (a[v'/x]) \Downarrow \langle u, cs, v\rangle\right) \qquad \langle f, _, _\rangle \notin cs}{\left(o \vdash (\mathbf{let}\, x = f(e);\ a) \Downarrow \langle u, \{\langle f, [\![e]\!]\ o, v'\rangle\} \cup cs, v\rangle\right)}$$

$$\text{ActionBind}\ \frac{\left(o \vdash (a[([\![e]\!]\ o)/x]) \Downarrow \langle u, cs, v\rangle\right)}{\left(o \vdash (\mathbf{let}\, x = e;\ a) \Downarrow \langle u, cs, v\rangle\right)}$$

$$\text{ActionTrue}\ \frac{\begin{array}{ccc}\left(o \vdash (a_T) \Downarrow \langle u_T, cs_T, v_T\rangle\right) & & \left(o \vdash (a[v_T/x]) \Downarrow \langle u, cs, v\rangle\right) \\ [\![e]\!]\ o & u_T \# u & cs_T \# cs\end{array}}{\left(o \vdash (\mathbf{let}\, x = \mathbf{if}\ (e)\ \mathbf{then}\ a_T\ \mathbf{else}\ a_F;\ a) \Downarrow \langle u_T \cup u, cs_T \cup cs, v\rangle\right)}$$

$$\text{ActionFalse}\ \frac{\begin{array}{ccc}\left(o \vdash (a_F) \Downarrow \langle u_F, cs_F, v_F\rangle\right) & & \left(o \vdash (a[v_F/x]) \Downarrow \langle u, cs, v\rangle\right) \\ \neg([\![e]\!]\ o) & u_F \# u & cs_F \# cs\end{array}}{\left(o \vdash (\mathbf{let}\, x = \mathbf{if}\ (e)\ \mathbf{then}\ a_T\ \mathbf{else}\ a_F;\ a) \Downarrow \langle u_F \cup u, cs_F \cup cs, v\rangle\right)}$$

$$\text{ActionAssert}\ \frac{[\![e]\!]\ o \qquad \left(o \vdash (a) \Downarrow \langle u, cs, v\rangle\right)}{\left(o \vdash (\mathbf{when}(e) \Rightarrow\ a) \Downarrow \langle u, cs, v\rangle\right)}$$

$$\text{ActionReturn}\ \frac{}{\left(o \vdash (\mathbf{return}(e)) \Downarrow \langle \{\}, \{\}, [\![e]\!]\ o\rangle\right)}$$

In the *ActionCall* inference rule, the value $v'$ is returned by the method $f$ on supplying the parameter ($[\![e]\!]\ o$). As discussed above, it is an unconstrained free variable in this inference rule. The *ActionTrue* and *ActionFalse* inference rules only execute the appropriate action on the *if*-side or the *else*-side, respectively, depending on the evaluation of the conditional predicate $e$. Similarly, the *ActionAssert* inference rule executes the action only when the guard $e$ is evaluated to True.

We are now in a position to formally give the modular semantics of a module.

### 6.3.3 Semantics for Modules

In this section and the next, we will give *modular semantics* for modules. The first question that comes to mind is why we need to offer semantics different from the inlining semantics described in the previous section. To answer this question, consider a hardware design specification consisting of two modules $A$ and $B$. Let module $A+B$ represent the overall specification of the full system. Let us say, we refine module $A$ to obtain module $A'$. If we prove that $A'$ actually refines $A$, we want to be able to conclude that $A' + B$ refines $A + B$ without having to prove this *ab initio*. Hence the need for modular semantics to define the behavior of $A$ and $A'$ independently of the module they are connected to.

Of course, the semantics of a composition of modules should match the semantics of the inlined version of the composition in order for it to be acceptable; the inlined version of a module is essentially the intuitive meaning of the module, and therefore any modular semantics should produce the same behavior as an inlined version of the module. We will later show that this is indeed the case (Lemma 2).

The semantics of a module is defined in the following:

1. *Step* representing the action performed by a collection of methods and at most one rule in a module. This step represents either a set of methods that can be called from another module, or a rule executed by the current module. To illustrate the meaning of this combination step, consider a rule in one primitive module calling some methods of another primitive module. Then the steps representing the rule of the first is combined with the steps representing the set of methods of the second module (and the called methods are hidden from further calls).

2. *Multistep* representing a sequence of module steps.

### Step Semantics of a Module

Each judgment for the step of a module $m$ is of the form $\langle m, o\rangle \Downarrow \langle u, \langle\alpha, ds, cs\rangle\rangle$ which we read as a *step of a module* $m$ *from state* $o$ *to state* $o[u]$, *executed by* $\textbf{\textit{Rule}}\langle k\rangle$ *(if* $\alpha = \textbf{\textit{Rule}}\langle k\rangle$*), and by methods* $ds$ *(if* $ds \neq \{\}$*), calling methods* $cs$; $o[u]$ represents the state obtained by applying the updates $u$ on state $o$. We call $o$ the *starting state*, $u$ the *updates*, $ds$ the *defined-methods set* of the step, $cs$ the *called-methods set*, $o[u]$ the *final state* or the *ending state*, and $\alpha$ the annotation of the step. We call a step executed by a rule a *rule step* and one executed by a method a *method step.*

We call $\langle\alpha, ds, cs\rangle$ the label generated by the step. If $\ell = \langle\alpha, ds, cs\rangle$, then the functions ($\mathsf{annotate}\ \ell$), ($\mathsf{defs}\ \ell$) and ($\mathsf{calls}\ \ell$) returns $\alpha$, $ds$ and $cs$, respectively.

We combine two steps of primitive modules as long as they have the same starting states, and have disjoint updates, called-methods sets and defined-methods sets. Moreover, both of the steps cannot be annotated as a rule. This is given by the predicate $\mathsf{canCombine}\ \langle o_1, u_1, \ell_1\rangle\ \langle o_2, u_2, \ell_2\rangle$, where $\langle o_1, u_1, \ell_1\rangle$ and $\langle o_2, u_2, \ell_2\rangle$ represent two primitive steps $\langle m, o_1\rangle \Downarrow \langle u_1, \ell_1\rangle$ and $\langle m, o_2\rangle \Downarrow \langle u_2, \ell_2\rangle$, respectively. For convenience, we first define another predicate $\mathsf{notBothRule}$ which ensures that two labels are not both annotated as rules; we overload it to work on annotations instead of labels.

**Definition 19.**

$$\begin{aligned}\mathsf{notBothRule}\ \alpha_1\ \alpha_2 &\triangleq \alpha_1 = \textbf{Meth} \lor \alpha_2 = \textbf{Meth}\\ \mathsf{notBothRule}\ \ell_1\ \ell_2 &\triangleq \mathsf{notBothRule}\ (\mathsf{annotate}\ \ell_1)\ (\mathsf{annotate}\ \ell_2)\end{aligned}$$

**Definition 20.** We say that two steps of a module, one starting with state $o_1$, producing updates $u_1$ and label $\ell_1$ and the other, starting with state $o_2$, producing updates

$u_2$ and label $\ell_2$ *can be combined* iff

$$\begin{aligned}\mathsf{canCombine}\ \langle o_1, u_1, \ell_1\rangle\ \langle o_2, u_2, \ell_2\rangle \triangleq\ & o_1 = o_2 \\ & \wedge u_1 \# u_2 \\ & \wedge ds_1 \# ds_2 \\ & \wedge cs_1 \# cs_2 \\ & \wedge (\mathsf{notBothRule}\ \ell_1\ \ell_2)\end{aligned}$$

For convenience, we define an operation $\ell_1 \oplus \ell_2$ to combine two labels as follows. We call this operation *combining labels $\ell_1$ and $\ell_2$*. We overload the $\oplus$ operator to work on annotations in addition to full labels.

**Definition 21.**

$$\begin{aligned}\alpha_1 \oplus \alpha_2 &\triangleq \textbf{if}\ (\alpha_1 = \textbf{Meth})\ \textbf{then}\ \alpha_2\ \textbf{else}\ \alpha_1 \\ \ell_1 \oplus \ell_2 &\triangleq \langle \mathsf{annotate}\ \ell_1 \oplus \mathsf{annotate}\ \ell_2, \mathsf{defs}\ \ell_1 \cup \mathsf{defs}\ \ell_2, \mathsf{calls}\ \ell_1 \cup \mathsf{calls}\ \ell_2\rangle\end{aligned}$$

When we are composing two modules, we need to take care of one more issue. We have to ensure that whenever a method is present in both the defined-methods set and the called-methods set in the label of a step, then the values of the arguments and return values must match, respectively, for the elements in the defined-methods set and the called-methods set. We implicitly ensure this as follows.

First, we *hide* the methods in the defined-methods set of the label which are also in the called-methods set of the label, and hiding the methods in the called-methods set of the label which are also in the defined-methods set. Function **hide** performs this operation. We call the operation of applying **hide** on a label as *hiding a label*, and the resulting label as a *hidden label*.

**Definition 22.** $\mathsf{hide}\ \langle \alpha, ds, cs\rangle \triangleq \langle \alpha, ds \setminus cs, cs \setminus ds\rangle$

Next, we ensure that the methods in the called-methods set in the final label are

not also in the static set of defined methods of the module (*i.e.,* the set returned by applying function **getDefs** defined in Section 6.1) and the methods in the defined-methods set in the final label are not in the static set of called-methods (*i.e.,* the set returned by applying function **getCalls** defined in Section 6.1) . Predicate wellHidden checks for this condition. Whenever (wellHidden $\ell$ $m$), we say that $\ell$ *is* wellHidden *w.r.t. module* $m$.

**Definition 23.** wellHidden $\langle \alpha, ds, cs \rangle \; m \triangleq (ds \# \mathsf{calls}\; m) \wedge (cs \# \mathsf{defs}\; m)$

Hiding a label obtained from a judgment for step and then checking the wellHidden condition on the final label ensures that the methods common to both the defined-methods set and the called-methods set in the labels have the same arguments and return values. This is given by the following theorem.

**Lemma 1.** Whenever the result of hiding a label produced by a step of a module is wellHidden with respect to the module, if a method is present in both the defined-methods set and called-methods set of the label, then it has the same argument and same return value in both these sets.

$$\begin{aligned} \forall m\; o\; u\; \ell\; a\; v\; a'\; v'.\;\; & \langle m, o\rangle \Downarrow \langle u, \ell\rangle \\ & \wedge\; \mathsf{wellHidden}\; (\mathsf{hide}\; \ell)\; m \\ & \wedge\; (\langle f, a, v\rangle \in \mathsf{defs}\; \ell) \\ & \wedge\; (\langle f, a', v'\rangle \in \mathsf{calls}\; \ell) \\ & \Rightarrow \langle a, v\rangle = \langle a', v'\rangle \end{aligned}$$

*Proof.* This follows because the methods in the defined-methods set of a label are distinct. Since hide removes the matching defined methods, if there are any defined methods left after applying hide because the argument or return values did not match, then they will be part of the static set of called methods of the module, which cannot be the case because of the assumption. A similar argument holds for the methods in the called-methods set of the label. □

It is useful to define another predicate combining the actions of combining two labels, hiding the resulting label and checking if the resulting label is well hidden w.r.t. a module as follows:

**Definition 24.** We say that *labels $\ell_1$ and $\ell_2$ can be well composed with respect to module $m$* if the following holds.

$$\begin{aligned}\mathsf{wellComposed}\ \ell_1\ \ell_2\ m \triangleq\ & \mathsf{notBothRule}\ \ell_1\ \ell_2 \\ & \wedge \mathsf{wellHidden}\ (\mathsf{hide}\ (\ell_1 \oplus \ell_2))\ m\end{aligned}$$

A step is given by the following inference rules.

**Definition 25** (Step).

$$\text{StepRule}\ \frac{o \in \mathsf{domain}\ \mathit{regs} \qquad \left(o \vdash (a) \Downarrow \langle u, cs, v\rangle\right) \qquad \langle k, a\rangle \in \mathit{rules}}{\langle\langle \mathit{regs}, \mathit{rules}, \mathit{meths}\rangle, o\rangle \Downarrow \langle u, \langle \mathbf{Rule}\langle k\rangle, \{\}, cs\rangle\rangle}$$

$$\text{StepMethod}\ \frac{o \in \mathsf{domain}\ \mathit{regs} \qquad \left(o \vdash (a[y/x]) \Downarrow \langle u, cs, v\rangle\right) \qquad \langle f, \lambda x.a\rangle \in \mathit{meths}}{\langle\langle \mathit{regs}, \mathit{rules}, \mathit{meths}\rangle, o\rangle \Downarrow \langle u, \langle \mathbf{Meth}, \{\langle f, y, v\rangle\}, cs\rangle\rangle}$$

$$\text{StepEmptyRule}\ \frac{o \in \mathsf{domain}\ \mathit{regs}}{\langle\langle \mathit{regs}, \mathit{rules}, \mathit{meths}\rangle, o\rangle \Downarrow \langle \{\}, \langle \mathbf{Rule}\langle \epsilon\rangle, \{\}, \{\}\rangle\rangle}$$

$$\text{StepEmptyMethod}\ \frac{o \in \mathsf{domain}\ \mathit{regs}}{\langle\langle \mathit{regs}, \mathit{rules}, \mathit{meths}\rangle, o\rangle \Downarrow \langle \{\}, \langle \mathbf{Meth}, \{\}, \{\}\rangle\rangle}$$

$$\text{StepUnion}\ \frac{\begin{array}{c}m = \langle \mathit{regs}, \mathit{rules}, \mathit{meths}\rangle \qquad \langle m, o\rangle \Downarrow \langle u, \ell\rangle \qquad \langle m, o\rangle \Downarrow \langle u', \ell'\rangle \\ \mathsf{canCombine}\ \langle o, u, \ell\rangle\ \langle o, u', \ell'\rangle\end{array}}{\langle m, o\rangle \Downarrow \langle u \cup u', \ell \oplus \ell'\rangle}$$

$$\text{StepComposition}\ \frac{\begin{array}{c}\langle m_1, o_1\rangle \Downarrow \langle u_1, \ell_1\rangle \qquad \langle m_2, o_2\rangle \Downarrow \langle u_2, \ell_2\rangle \\ \mathsf{wellComposed}\ \ell_1\ \ell_2\ (m_1 + m_2)\end{array}}{\langle m_1 + m_2, o_1 \cup o_2\rangle \Downarrow \langle u_1 \cup u_2, \mathsf{hide}\ (\ell_1 \oplus \ell_2)\rangle}$$

Each rule and method of a primitive module leads to a step of the primitive module. We also have *empty-rule step* and an *empty-method step* for primitive modules. Both the empty-rule and empty-method steps do not perform any state updates nor

call any methods. The empty-rule step is annotated with **Rule**$\langle\epsilon\rangle$ and the empty-method step is annotated with **Meth**.

The empty-method step is just like a "nil" constructor for a list. The need for the empty-rule step is subtle and will become clear once we define the notion of refinement formally. Informally the reason is as follows: in order to claim that a module $A$ implements another module $B$, any step sequence in $A$ should have the same external behavior as some step sequence in $B$. But if $A$'s step sequence involves an execution of a rule which does not call any external methods and does not produce any state changes, this particular step can effectively be "simulated" by not doing any steps in $B$. However, $A$ does not permit the execution of any other rule anywhere else in the modules connected to $A$ since $A$ has executed a rule. In order to have a similar constraint on the modules connected to $B$, $B$ must also be a step executed by a rule, which will be modeled by the empty-rule step.

The reason for creating a step combining several methods in a primitive module is straightforward: we want this step to be combinable with another step of another module. Say the other module's step is executed by a rule and that rule calls several methods of the first module. Then, the first module has to expose a step containing several methods for it to be combinable with the second module's step executed by the rule.

The reason for allowing a rule to be combined with several methods in a step is more complicated. Consider a scenario where a rule of one module calls a method defined in another module which in turns calls a method in the first module. This is a perfectly legal composition of modules as there are no cycles in the call graph. In general, an external method call can *transitively* call a method defined by a module as long as there are no cycles in the call graph. In such cases, we need to have a judgment in the first module which performs the actions of the rule as well as the method that external modules call. In general, since we do not know which

combinations of methods of a module will be called simultaneously externally, we need to create judgments containing every possible combination of methods with a rule.

In all the steps of primitive modules, we have to ensure that the domain of registers in the starting state is the same of the registers defined in the primitive module.

Finally, the step representing the composition of two modules ensures the following:

1. The label of at least one of the modules is annotated as a method.

2. The final label contains the union of the defined-methods sets of the two modules and the union of their called-methods sets, except that the final defined-methods set does not contain the static set of called methods of the composed module and the final called-methods set does not contain the static set of defined methods of the composed module (checked using $\mathsf{wellHidden}$ predicate).

3. Whenever a method is called by one module and defined in the other,

   (a) the values of the arguments and return values match, respectively; and

   (b) the element corresponding to that method is removed from the defined-methods set of the first module and called-methods set of the other (using the $\mathsf{hide}$ function).

It is straightforward to see that the step semantics of any module are consistent with the step semantics of the inlined version of the module (Section 6.2), as stated by the following theorem:

**Lemma 2.** A step of a module from state $o$ can produce updates $u$ and label $\ell$ iff there is a step of the inlined version of the module from state $o$ that can produce the

same updates $u$ and the same label $\ell$.

$$\forall m\ o\ u\ \ell.\ \langle m, o\rangle \Downarrow \langle u, \ell\rangle \Leftrightarrow \langle [\![m]\!], o\rangle \Downarrow \langle u, \ell\rangle$$

### Multistep Semantics for Modules

Each judgment for the multistep semantics of a module $m$ is of the form $\left(o \xrightarrow[m]{\sigma}\!\!\!\!\rightarrow o'\right)$, which we read as a *multistep of module $m$ starting from state $o$ reaching state $o'$, producing a label sequence $\sigma$*. We call $o$ the *starting state*, $o'$ the *ending state* or *final state* and $\sigma$ the *label sequence* for the multistep.

**Definition 26** (Multistep).

$$\text{Nil}\frac{}{\left(o \xrightarrow[m]{[]}\!\!\!\!\rightarrow o\right)}$$

$$\text{Multi}\frac{\left(o \xrightarrow[m]{\sigma}\!\!\!\!\rightarrow o'\right) \qquad \langle m, o'\rangle \Downarrow \langle u, \ell\rangle}{\left(o \xrightarrow[m]{\ell::\sigma}\!\!\!\!\rightarrow o'[u]\right)}$$

We define the *behavior of a module* as a multistep starting from the initial state of the module (given by the **getRegs** function defined in Section 6.1). We read $(m \Downarrow \langle o, \sigma\rangle)$, as *module $m$ reaching a state $o$ and producing a label sequence $\sigma$*.

**Definition 27.**

$$\text{Behavior}\frac{\left((\mathsf{getRegs}\ m) \xrightarrow[m]{\sigma}\!\!\!\!\rightarrow o\right)}{(m \Downarrow \langle o, \sigma\rangle)}$$

We present a theorem to split a behavior in a composition of two modules into two behaviors, one for each module. We use $(\sigma!i)$ to denote the $i^{\text{th}}$ element in the list $\sigma$ and $|\sigma|$ to denote the length of the list.

**Theorem 1.** A behavior of a module composed of two modules can be split into individual behaviors of the two modules such that

1. the sizes of the label sequences produced by the three modules are the same;

and

2. the $i^{\text{th}}$ labels in the sequences of the two modules can be well composed to form the $i^{\text{th}}$ label of the sequence of the composed module.

That is,

$$
\begin{aligned}
&\forall m_1\ m_2\ o\ \sigma.\ (m_1 + m_2 \Downarrow \langle o, \sigma \rangle) \\
&\qquad \Leftrightarrow \exists o_1\ o_2\ \sigma_1\ \sigma_2.\ (m_1 \Downarrow \langle o_1, \sigma_1 \rangle) \\
&\qquad\qquad\qquad \wedge\ (m_2 \Downarrow \langle o_2, \sigma_2 \rangle) \\
&\qquad\qquad\qquad \wedge\ |\sigma| = |\sigma_1| = |\sigma_2| \\
&\qquad\qquad\qquad \wedge\ \forall i < |\sigma|.\ \mathsf{wellComposed}\ (\sigma_1!i)\ (\sigma_2!i)\ (\sigma!i)
\end{aligned}
$$

*Proof.* This follows straightaway by induction on a behavior, and by applying the inference rule *StepComposition* on each step. □

## 6.4 Trace-Refines Relation

We need a notion of when one module *implements* another. Informally, a module that has the same interactions with the environment as another module can be considered as a safe substitute for the latter. We will define an asymmetrical notion of compatibility.

We relax the notion of refinement such that the interactions of an implementation with the external environment need not exactly match that of the specification. We need to allow such a relaxed definition in order to enable modular proofs. In fact, in Section 6.5, we will give a theorem which shows that when a pair of modules refine another pair of modules using our relaxed notion of refinement, then the composition of the former pair *exactly* refines the composition of the latter pair (Theorem 7). Such a notion is necessary in our proof for the complex multiprocessor system implementing

sequential consistency. The speculative processor $P_{\text{so}}$ sends speculative load requests to the memory and receives speculative load responses from the memory, while there is no equivalent for these requests and responses in the decoupled processor $P_{\text{dec}}$. Ignoring these speculative load requests and responses, the speculative processor does indeed *behave similarly* to the decoupled processor.

In order to account for the relaxed notion of the implements relation between two modules we use a function $p : (\mathcal{F} \times \mathcal{C} \times \mathcal{C}) \rightarrow (\mathcal{C} \times \mathcal{C})^{\epsilon}$. Function $p$ converts a tuple $\langle f, a, r\rangle$ into another tuple $\langle f, a', r'\rangle$ or removes it (*i.e.,* maps it to $\epsilon$) and hence is called a *label map*. Label map $p$ is used to alter the methods in the defined-methods set and the called-methods set of the labels produced by the implementation. We transform the function $p$ which operates on $\mathcal{F} \times \mathcal{C} \times \mathcal{C}$ into one that operates on a label as shown below. Function $\hat{p}$ applies function $p$ on every defined-methods set and called-methods set of the label, and keeps only those values that are not $\epsilon$ in the respective sets. It also removes the name of the rule in a rule-annotated label.

**Definition 28.**

$$\begin{aligned}\hat{p}\ \langle x, ds, cs\rangle \triangleq\quad & \langle \textbf{if } (x = \textbf{Meth}) \textbf{ then Meth else Rule}, \\ & \{\langle f, a', r'\rangle | \langle f, a, r\rangle \in ds, p\ \langle f, a, r\rangle = \langle a', r'\rangle\}, \\ & \{\langle f, a', r'\rangle | \langle f, a, r\rangle \in cs, p\ \langle f, a, r\rangle = \langle a', r'\rangle\}\rangle\end{aligned}$$

We use *p-transformation* to denote the label transformed by applying the label map $p$ on all the called and defined methods along with their arguments and return values, *i.e.,* $p$-transformation of a label $\ell = \hat{p}\ \ell$. We say $\ell_1$ is $\mathsf{indistinguishable}$ from $\ell_2$ w.r.t. label map $p$ iff the $\mathsf{id}$-transformation of $\ell_1$ matches the $p$-transformation of $\ell_2$, where $\mathsf{id}$ represents an identity label map. We overload the term to denote two label sequences to be $\mathsf{indistinguishable}$ as follows:

**Definition 29.**

$$\begin{aligned}\textsf{indistinguishable}\ \ell_1\ \ell_2\ p &\triangleq \hat{\textsf{id}}\ \ell_1 = \hat{p}\ \ell_2 \\ \textsf{indistinguishable}\ \sigma_1\ \sigma_2\ p &\triangleq (\textsf{map}\ \hat{\textsf{id}}\ \sigma_1) = (\textsf{map}\ \hat{p}\ \sigma_2)\end{aligned}$$

We are now in a position to define the *implements* or the *trace-refines* relation.

**Definition 30.** Let $p : \mathcal{L} \to \mathcal{L}$ be a label map which is able to replace called or defined method names with arguments and return values with alternative names or values, or erase them altogether. Let $m_1$ and $m_2$ be two modules. We say that $m_1$ **trace-refines** $m_2$ **w.r.t.** $p$ or $m_1$ $p$**-implements** $m_2$, denoted as $m_1 \sqsubseteq_p m_2$, iff for every label sequence $\sigma_1$ produced by module $m_1$, module $m_2$ reaches a state $s_2$ producing a label sequence $\sigma_2$ which is $\textsf{indistinguishable}$ from $\sigma_1$ w.r.t. label map $p$. That is,

$$\begin{aligned}m_1 \sqsubseteq_p m_2 \triangleq \forall s_1\ \sigma_1.\ (m_1 \Downarrow \langle s_1, \sigma_1 \rangle) \\ \Rightarrow \exists s_2\ \sigma_2.\ (m_2 \Downarrow \langle s_2, \sigma_2 \rangle) \\ \wedge (\textsf{map}\ \hat{p}\ \sigma_1) = (\textsf{map}\ \hat{\textsf{id}}\ \sigma_2)\end{aligned}$$

As a shorthand, we write $m_1 \sqsubseteq m_2$ for $m_1 \sqsubseteq_{\textsf{id}} m_2$, for $\textsf{id}$ an identity label map and in that case, we say $m_1$ implements $m_2$.

We can also define a notion of equivalence between two modules, denoted as $m_1 \equiv m_2$ as follows, which means that the two modules implement each other. Under this notion, we say that $m_1$ *is equivalent to* $m_2$.

**Definition 31.** $m_1 \equiv m_2 \triangleq m_1 \sqsubseteq m_2 \wedge m_2 \sqsubseteq m_1$

## 6.5 Properties of Trace-Refines Relation

It is easy to see that the following properties hold for the trace-refines relation:

**Theorem 2** (Reflexivity). $\forall m.\ m \equiv m$

**Theorem 3** (Transitivity).

$$\begin{aligned} \forall m_1\, m_2\, m_3\, p\, q.\ & m_1 \sqsubseteq_p m_2 \\ & \wedge\ m_2 \sqsubseteq_q m_3 \\ & \Rightarrow\ m_1 \sqsubseteq_{q \circ p} m_3 \end{aligned}$$

*Proofs of Theorem 2 and Theorem 3.* The above two theorems can easily be proved using just the definition of trace refinement. □

Next, we show the refinement relationship between a module and its inlined counterpart.

**Theorem 4.** A module is equivalent to the inlined version of itself.

$$\forall m.\ m \equiv [\![m]\!]$$

*Proof.* This follows using induction on a behavior, since for every step of $m$, there is a corresponding step in the inlined version of $m$ (and vice versa) according to Theorem 2. □

**Theorem 5** (Commutativity and Associativity).

1. $\forall m_1\, m_2.\ m_1 + m_2 \equiv m_2 + m_1$
2. $\forall m_1\, m_2\, m_3.\ (m_1 + m_2) + m_3 \equiv m_1 + (m_2 + m_3)$

*Proof.* Both these properties can be proved directly using Theorem 1 since the $\oplus$ operator is both commutative and associative. □

It is useful to talk about individual refinements of two non-interacting modules. When no method that is defined by one module is called by the other, we say that the two methods are *non-interacting.* This notion can be formally defined using the $\mathsf{nonInteracting}$ predicate.

**Definition 32.**

$$\begin{aligned}\mathsf{nonInteracting}\ m_1\ m_2 \triangleq\ & \mathsf{getDefs}\ m_1 \# \mathsf{getCalls}\ m_2 \\ & \wedge \mathsf{getDefs}\ m_2 \# \mathsf{getCalls}\ m_1\end{aligned}$$

We also need to ensure that the label map $p$ does not map two different methods to the same method. This is given precisely by the $\mathsf{uniqMap}$ predicate. We say that the label map $p$ *performs a unique mapping* if the following holds.

**Definition 33.**

$$\begin{aligned}\mathsf{uniqMap}\ p \triangleq \forall f_1\ a_1\ r_1\ f_2\ a_2\ r_2\ f\ a'_1\ r'_1\ a'_2\ r'_2.\ & p\ \langle f_1, a_1, r_1\rangle = \langle f, a'_1, r'_1\rangle \\ \wedge\ & p\ \langle f_2, a_2, r_2\rangle = \langle f, a'_2, r'_2\rangle \\ \Rightarrow\ & f_1 = f_2\end{aligned}$$

This gives us the theorem about refinement of composition of non-interacting modules.

**Theorem 6** (Trace-Refines Relation for Non-Interacting Composition)**.** Given a label map $p$ which performs a unique mapping, if $m_1$ and $m_2$ $p$-implement $m'_1$ and $m'_2$, respectively, and $m_1$ and $m_2$ are non-interacting and so are $m'_1$ and $m'_2$, then $m_1 + m_2$

$p$-implements $m'_1 + m'_2$.

$$\begin{aligned}\forall m_1\ m_2\ m'_1\ m'_2\ p.\ \ & m_1 \sqsubseteq_p m'_1 \\ & \wedge\ m_2 \sqsubseteq_p m'_2 \\ & \wedge\ \mathsf{uniqMap}\ p \\ & \wedge\ \mathsf{nonInteracting}\ m_1\ m_2 \\ & \wedge\ \mathsf{nonInteracting}\ m'_1\ m'_2 \\ & \Rightarrow m_1 + m_2 \sqsubseteq_p m'_1 + m'_2\end{aligned}$$

We now state a refinement property when combining interacting modules. Say there is a label map $p$ which operates only on the methods that are shared between two modules while leaving the rest of the methods as they are. Predicate transformationIsHidden captures this condition. We say that the *transformation of $p$ is hidden with respect to module $m$* if the following holds.

$$\begin{aligned}\mathsf{transformationIsHidden}\ p\ m \triangleq \forall f\ a\ r.\ & p\ \langle f, a, r\rangle \neq \langle a, r\rangle \\ & \Rightarrow\ f \in (\mathsf{getDefs}\ m \cap \mathsf{getCalls}\ m)\end{aligned}$$

In this case, the composition of the two modules results in an id-based implements relation, as shown below:

**Theorem 7** (Trace-Refines Relation for Interacting Composition). If $m_1$ and $m_2$ $p$-implement $m'_1$ and $m'_2$, respectively, and the transformation of $p$ is hidden with respect to modules $m_1$ and $m_2$, then $m_1 + m_2$ implements $m'_1 + m'_2$.

$$\begin{aligned}\forall m_1\ m_2\ m'_1\ m'_2\ p.\ & m_1 \sqsubseteq_p m'_1 \\ & \wedge\ m_2 \sqsubseteq_p m'_2 \\ & \wedge\ \mathsf{transformationIsHidden}\ p\ (m_1 + m_2) \\ & \Rightarrow m_1 + m_2 \sqsubseteq m'_1 + m'_2\end{aligned}$$

*Proofs of Theorem 6 and Theorem 7.* The proofs of both the above theorems rely on Theorem 1. Proving the non-interacting composition theorem is straightforward. The interacting composition theorem can be proved as follows. The main insight is that function **hide** ensures that all the common methods between the two modules are hidden in the combined label. So it does not matter if the common methods are transformed. To prove the theorem, the combined multistep for module $(m_1 + m_2)$ is split into two multisteps for modules $m_1$ and $m_2$ respectively. On each of their label sequences some of the common methods are mapped to other methods (or $\epsilon$). Finally the transformed label sequences are combined once again. This last combined label sequence is the same as that produced by the composition $(m_1 + m_2)$ because the transformed labels are hidden. □

### 6.5.1 Decomposing a Multistep

The trace-refines relation in Section 6.4 requires us to reason about multisteps, *i.e.*, sequences of steps. Verification becomes much easier if the reasoning is restricted to steps, and even more so if the reasoning is restricted to steps executed by at most a single rule or a single method – we call such steps unit steps.

The idea of breaking a refinement relation into a *simulation relation* for breaking the proof into manageable units is well known [7, 14, 49]. If there is a simulation relation between two transition systems, one being an implementation and the other being the specification, then the refinement relation can be proved between these systems using induction on the sequence of transitions of the implementation system. We will now state a similar theorem which allows us to conclude that a module refines another module based on the properties of the unit steps of the two modules.

The trace-refines relation only relates the label sequence of an implementation to that of the specification. While it does not require any relation between the state of

the implementation and that of the specification, we have to give a refinement map from a state of the implementation to that of the specification if we use a simulation-like relation on unit steps to derive a trace-refines relation on multisteps. We call such a refinement map a *state map*.

There is one major distinction between a step in our semantics and a transition in labeled transition systems. In labeled transition systems, any sequence of transitions is allowed as long as the starting state of one transition is the ending state of the previous transition. Once a simulation relation is established relating a single transition in one system to a sequence of zero or more transitions in another system, one can conclude that the former refines the latter. However, for us, a step in a primitive module is built using potentially several unit steps. The combination of steps for the primitive module all start from the same state. Moreover, we need to ensure that the combining steps write to disjoint sets of registers, and call disjoint sets of methods. Thus, in order to conclude that one module trace-refines another using the relation between the unit steps in the two modules, we need a condition which does not have a counterpart in labeled transition systems. We will formally describe the extra conditions for establishing the trace-refines relation below.

We formally define a unit step below using the isUnit predicate on the label of a step.

**Definition 34.** A step containing label $\ell$ is said to be a unit step iff the size of the defined-methods set is not greater than 1.

$$\mathsf{isUnit}\ \ell \triangleq |ds| \leq 1$$

Note that in the above definition, even empty-rule steps and empty-method steps are unit steps. We call a unit step executed by a rule a *unit rule step* and a unit step executed by a method a *unit method step*.

Let's say we have two modules $m_{\mathsf{impl}}$ and $m_{\mathsf{spec}}$. Say we have a state map $\theta : \mathcal{S} \rightarrow \mathcal{S}$, where $\mathcal{S}$ represents the set of register maps, *i.e.*, $\overrightarrow{\mathcal{R}\times\mathcal{C}}$. Function $\theta$ maps every possible state of $m_{\mathsf{impl}}$ (whether it is reachable or not) to a state in $m_{\mathsf{spec}}$. Given a label map $p$ transforming the elements of the defined-methods set and called-methods set, we are now going to give a strong sufficient condition to check whether $m_{\mathsf{impl}}$ $p$-implements $m_{\mathsf{spec}}$. We need to define some predicates before giving the main theorem. We also need a function $T : (\mathcal{S} \times \mathcal{S} \times \mathcal{L}) \rightarrow (\mathcal{S} \times \mathcal{S} \times \mathcal{L})$ which maps a 3-tuple $\langle o, u, \ell \rangle$ of a unit step $\langle m_{\mathsf{impl}}, o \rangle \Downarrow \langle u, \ell \rangle$ to a 3-tuple $\langle o', u', \ell' \rangle$ of a unit step $\langle m_{\mathsf{spec}}, o' \rangle \Downarrow \langle u', \ell' \rangle$. This function is used to obtain the exact unit step of the specification from the unit step of the implementation and hence called a unit step map.

Not all unit step maps $T$ are acceptable. We have to ensure that $T$ is consistent with state map $\theta$ and label map $p$ for the two modules $m_{\mathsf{impl}}$ and $m_{\mathsf{spec}}$. The exact conditions are shown in the definition of predicate $\mathsf{consistentSubstepMap}$ below.

**Definition 35.** A unit step map $T : (\mathcal{S} \times \mathcal{S} \times \mathcal{L}) \rightarrow (\mathcal{S} \times \mathcal{S} \times \mathcal{L})$ is *consistent w.r.t. a state map $\theta : \mathcal{S} \rightarrow \mathcal{S}$ and a label map $p : \mathcal{L} \rightarrow \mathcal{L}$ and modules $m_{\mathsf{impl}}$ and $m_{\mathsf{spec}}$,* iff for any state $o$ reached by $m_{\mathsf{impl}}$, if there is another unit step of $m_{\mathsf{impl}}$ starting with state $o$, producing updates $u$ and label $\ell$, there is a step of $m_{\mathsf{spec}}$ starting with state $o'$, producing updates $u'$ and label $\ell'$ such that $T\ \langle o, u, \ell \rangle = \langle o', u', \ell' \rangle$ and $\ell'$ is

$\mathsf{indistinguishable}$ from $\ell$ w.r.t. $p$. That is,

$$\begin{aligned}\mathsf{consistentSubstepMap}\ T\ m_{\mathsf{impl}}\ m_{\mathsf{spec}}\ \theta\ p \ \triangleq\ \forall o\ \ell\ u.\ & (m_{\mathsf{impl}} \Downarrow \langle o, _\rangle) \\ & \wedge\ \langle m_{\mathsf{impl}}, o\rangle \Downarrow \langle u, \ell\rangle \\ & \wedge\ \mathsf{isUnit}\ \ell \\ & \Rightarrow o' = \theta\ o \\ & \quad \wedge\ o'[u'] = \theta\ (o[u]) \\ & \quad \wedge\ \mathsf{indistinguishable}\ \ell'\ \ell\ p \\ & \quad \wedge\ \langle m_{\mathsf{spec}}, o'\rangle \Downarrow \langle u', \ell'\rangle \\ & \quad \textbf{where}\ \ \langle o', u', \ell'\rangle = T\ \langle o, u, \ell\rangle\end{aligned}$$

Whenever two unit steps of the implementation starting from some reachable state in the implementation can be combined, we also want the corresponding steps of the specification to be combinable. This is the extra condition that we had mentioned earlier compared to traditional simulation relations for transition systems. This is given by the predicate $\mathsf{specShouldCombine}$ as follows:

**Definition 36.** We say that the specification $m_{\mathsf{spec}}$ is *as combinable as the implementation w.r.t. unit step map $T$* if when two unit steps of an implementation $m_{\mathsf{impl}}$ starting from a reachable state can be combined, then the corresponding steps of the specification obtained by applying the unit step map on the two unit steps can also be combined. That is,

$$\begin{aligned}\mathsf{specShouldCombine}\ T\ m_{\mathsf{impl}} \ \triangleq\ \forall o\ \ell\ u\ \ell'\ u'.\ \ & (m_{\mathsf{impl}} \Downarrow \langle o, _\rangle) \\ & \wedge\ \langle m_{\mathsf{impl}}, o\rangle \Downarrow \langle u, \ell\rangle \\ & \wedge\ \langle m_{\mathsf{impl}}, o\rangle \Downarrow \langle u', \ell'\rangle \\ & \wedge\ \mathsf{canCombine}\ \langle o, u, \ell\rangle\ \langle o, u', \ell'\rangle \\ & \Rightarrow \mathsf{canCombine}\ (T\ \langle o, u, \ell\rangle)\ (T\ \langle o, u', \ell'\rangle)\end{aligned}$$

This gives us the main decomposition theorem as follows:

**Theorem 8** (Decomposition). Given modules $m_{\mathsf{impl}}$ and $m_{\mathsf{spec}}$, and a state map $\theta$, a label map $p$ and a unit step map $T$ from the state, labels and unit steps of $m_{\mathsf{impl}}$ to those of $m_{\mathsf{spec}}$, respectively, if $T$ is consistent w.r.t. $\theta$, $p$, $m_{\mathsf{impl}}$ and $m_{\mathsf{spec}}$ and $m_{\mathsf{spec}}$ is as combinable as $m_{\mathsf{impl}}$, then $m_{\mathsf{impl}}$ $p$-implements $m_{\mathsf{spec}}$. That is,

$$\begin{aligned}\forall m_{\mathsf{impl}}\ m_{\mathsf{spec}}\ p\ \theta\ T.\ & \mathsf{consistentSubstepMap}\ T\ m_{\mathsf{impl}}\ m_{\mathsf{spec}}\ \theta\ p \\ & \wedge\ \mathsf{specShouldCombine}\ T\ m_{\mathsf{impl}} \\ & \Rightarrow m_{\mathsf{impl}} \sqsubseteq_p m_{\mathsf{spec}}\end{aligned}$$

We say that *a state of the specification $o'$ is* **compatible** *with that of the implementation $o$ w.r.t. state map $\theta$ iff $o' = \theta\ o$*. Similarly, we say that *a label $\ell'$ of the specification is* **compatible** *with label $\ell$ of the implementation w.r.t. label map $p$ iff $\ell'$ is* **indistinguishable** *from $\ell$ w.r.t. $p$*. We also extend the **compatible** relation in the natural way from two labels to two sequences of labels. Finally, we say that a unit step $\langle m_{\mathsf{impl}}, o\rangle \Downarrow \langle u, \ell\rangle$ of the specification is **compatible** with a step $\langle m_{\mathsf{spec}}, o'\rangle \Downarrow \langle u', \ell'\rangle$ w.r.t unit step map $T$ iff $T\ \langle o, u, \ell\rangle = \langle o', u', \ell'\rangle$.

## 6.6 Weak Implements Relation

The definition of implements relation in Section 6.4 requires that the label sequence produced by the specification module must be **indistinguishable** from the label sequence produced by the refined module with respect to an **id** label map. While for the rest of the thesis, it is enough to use this notion of the implements relation, there are practical situations where this definition is too strict. Consider, for example, a scenario where we have two modules $m_1$ and $m_2$, and $m_2$ is similar to $m_1$ except that it executes multiple rules of $m_2$ simultaneously. This can happen if $m_2$ merges two distinct rules of $m_1$ into a single rule. The only distinction that the external environment can

observe between the two systems is that instead of observing two consecutive labels containing the methods called by the rules of $m_1$, it observes a single label because of the execution of the merged rule in $m_2$, which contains the union of the methods called by the individual rules of $m_1$.

With this intuition, we provide the semantics for the *weak-implements* relation between two modules as follows. We need to relax the constraint on the label sequences being indistinguishable w.r.t. id and instead define a congruence relation between two label sequences which permits merging of adjacent labels. We define such a relation ($\cong$) inductively below.

**Definition 37.**

$$\text{EqualEquiv}\frac{}{\sigma \cong \sigma}$$

$$\text{Merge}\frac{\mathsf{calls}\ \ell \ \#\ \mathsf{calls}\ \ell' \qquad \mathsf{defs}\ \ell \ \#\ \mathsf{defs}\ \ell'}{\sigma \mathbin{++} (\ell :: (\ell' \mathbin{++} \sigma')) \cong \sigma \mathbin{++} ((\ell \oplus \ell') :: \sigma')}$$

Any two sequences $\sigma_1$ and $\sigma_2$ are said to be congruent iff $(\mathsf{map}\ \hat{\mathsf{id}}\ \sigma_1) \cong (\mathsf{map}\ \hat{\mathsf{id}}\ \sigma_2)$. Using this definition of congruence of two label sequences, we can define the *weak-implements* relation as follows:

**Definition 38.** We say that $m_1$ **weak-implements** $m_2$, denoted as $m_1 \subseteq m_2$ iff for every label sequence $\sigma_1$ produced by module $m_1$, module $m_2$ reaches a state $s_2$ producing a label sequence $\sigma_2$ such that $\sigma_1$ and $\sigma_2$ are congruent. This is shown below:

$$\begin{aligned} m_1 \subseteq m_2 \triangleq \forall s_1\, \sigma_1.\ & (m_1 \Downarrow \langle s_1, \sigma_1 \rangle) \\ \Rightarrow\ & \exists s_2\, \sigma_2.\ (m_2 \Downarrow \langle s_2, \sigma_2 \rangle) \\ & \qquad \wedge (\mathsf{map}\ \hat{\mathsf{id}}\ \sigma_1) \cong (\mathsf{map}\ \hat{\mathsf{id}}\ \sigma_2) \end{aligned}$$

## 6.7 Limitations of the Semantics

The formal semantics is supposed to capture the intuitive notion of equivalence between two programs (or refinement, in our case). It is not very satisfactory if it creates

```
Module:
Regs y(False);
-----------------
Rule k1:
y := True;

Meth f():
return y;
```

(a) Module $m_1$

```
Module:
Regs x(False), y(False);
-----------------
Rule k2:
x := False;
y := True;

Meth f():
x := True;
return y;
```

(b) Module $m_2$

Figure 6-2: Modules $m_1$ and $m_2$ showing the limitations of our modular semantics

distinctions between two programs, one of which intuitively refines the other.

The modular semantics that we present for Bluespec almost captures every intuitive aspect of refinement except for one detail. Whenever a rule in a module is "closed", *i.e.,* it does not call any external methods, then a step in the module involving that unit rule step cannot be combined with any unit method steps. However, in our semantics we allow this combination. The following example illustrates the point. Consider two modules $m_1$ and $m_2$ shown in Figure 6-2.

In the above two examples, $m_1 \not\equiv m_2$ according to our semantics. There can be a step of $m_1$ with a label $\ell_1 = \langle \mathbf{Rule}\langle k_1 \rangle, \{\langle f, (), _\rangle\}, \{\}\rangle$, while there cannot be any step of $m_2$ with such a label since rule $r_2$ and method $f_2$ write to the same register $(x)$. However, since $k_1$ does not call any external methods, method $f$ in module $m_1$ can never be called from any external module. For any module $m$, $m_1 + m$ will behave exactly in the same manner as a composition $m_2 + m$. So, intuitively, $m_1$ is equivalent to $m_2$ while our semantics will show a distinction between the two modules.

In order to prevent such a label from being produced in a step of $m_1$, we would have to (a) separate the called-methods set of each unit step in a step; (b) allow a unit rule step to be combined with a set of unit method steps which contains at least one method in its defined-methods set only if the called-methods set of the unit rule step is non-empty.

The reason why we did not follow this procedure in the semantics is to simplify

the semantics. Since we are using these semantics to formally verify hardware designs in Coq, it is important to keep the semantics as simple as possible. This was a design decision we took since such arcane examples rarely show up in real hardware verification.

## 6.8 Conclusion

In this chapter we gave the semantics of Bluespec programs. We formalized the notion of implementation or refinement and stated a few useful properties about refinement. In the next two chapters, we will be using the definitions of refinement and its properties to prove that a multiprocessor system connected to a hierarchy of coherent caches implements the sequential consistency specification.

# Chapter 7

# Verifying the Complete Multiprocessor System Implementation

In this chapter we will prove the correctness of the full system; that is, we will show that the system consisting of speculative out-of-order processors connected to a hierarchy of coherent caches implements sequential consistency.

Figure 7-1 gives the overall structure of the proof. Formally, we prove $P^1_{\mathsf{so}} + \ldots + P^n_{\mathsf{so}} + M_{\mathsf{cache}} \sqsubseteq SC$ shown at the root of the overall proof, colored red. We start with the trace-refines relations for $P_{\mathsf{so}}$ cores and the cache-coherent memory that we will prove in Section 7.2 and in Chapter 8, respectively. From Theorem 10, we get $P_{\mathsf{so}} \sqsubseteq_{\mathsf{noSpec}} P_{\mathsf{dec}}$. Label map noSpec removes the method calls which enqueue speculative loads *i.e.*,,

**Definition 39.**

$$\mathsf{noSpec}\langle f, a, r\rangle = \begin{cases} \epsilon & : \langle f, a, r\rangle = \langle cRqs.enq, \langle \mathsf{Ld}, a', t\rangle, ()\rangle, t \neq \epsilon \\ \langle a, r\rangle & : \text{otherwise} \end{cases}$$

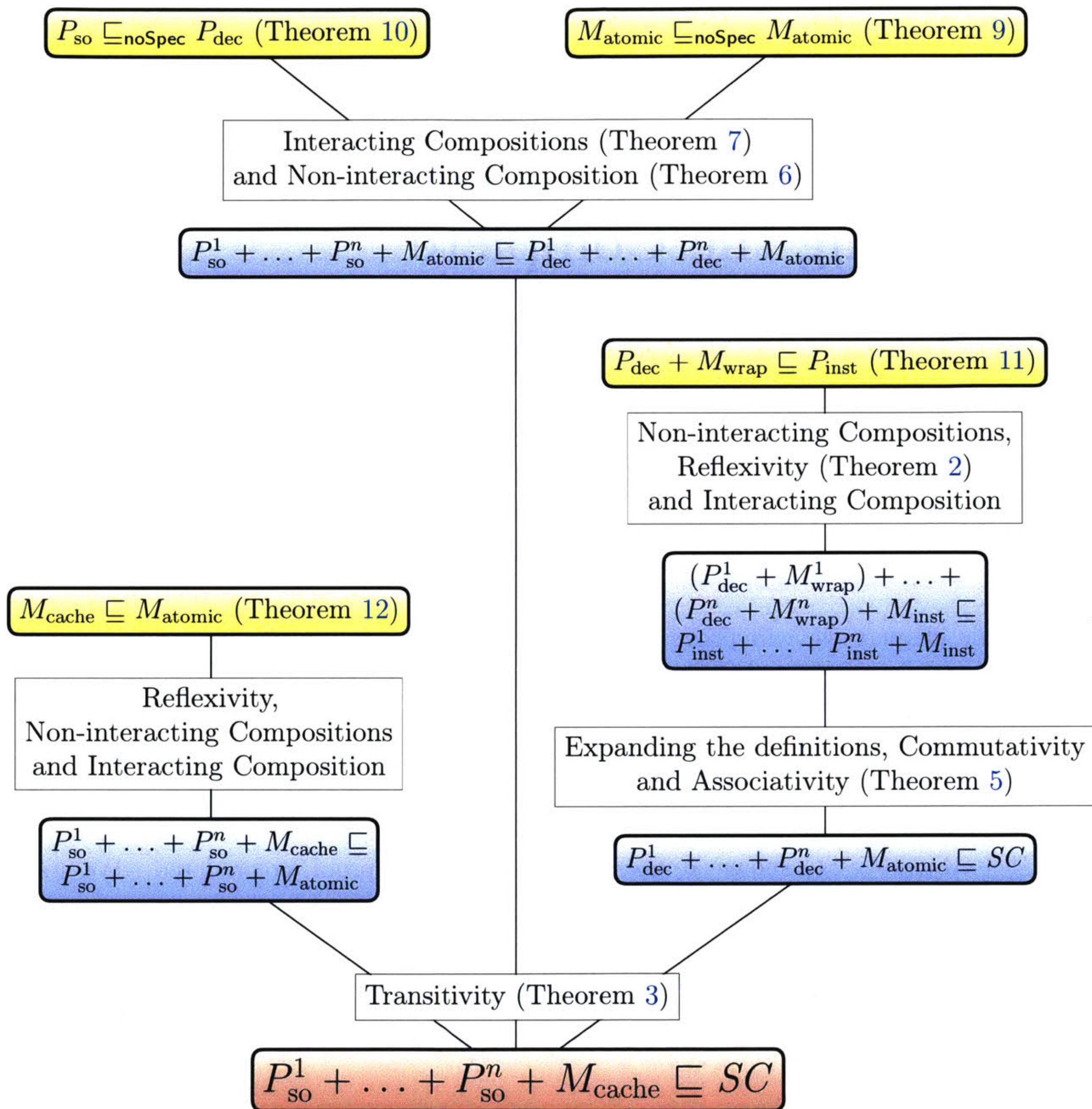



Figure 7-1: Overall structure of the proof of the full system

From Theorem 11, we get $P_{\text{dec}} + M_{\text{wrap}} \sqsubseteq P_{\text{inst}}$ and from Theorem 12 in the next chapter, we get $M_{\text{cache}} \sqsubseteq M_{\text{atomic}}$. These theorems are at the leaves of the overall proof and are colored yellow. In addition, we need to prove another property $M_{\text{atomic}} \sqsubseteq_{\text{noSpec}} M_{\text{atomic}}$, *i.e.,* whenever the requests and responses for speculative loads are dropped, the leftover label sequence can also be generated by $M_{\text{atomic}}$. We prove this in Section 9.

Using $M_{\text{cache}} \sqsubseteq M_{\text{atomic}}$, we can prove that a system composed of speculative out-of-order cores attached to a cache-coherent memory implements a system composed of the sames cores attached to a atomic memory. We need to first use reflexivity of implement relative (Theorem 2), thus giving us $P_{\text{so}} \sqsubseteq P_{\text{so}}$. Repeated application of trace-refines relation for non-interacting compositions gives us $P^1_{\text{so}} + \ldots + P^n_{\text{so}} \sqsubseteq P^1_{\text{so}} + \ldots + P^n_{\text{so}}$. Finally, the trace-refines relation for interacting compositions (Theorem 7) allows us to prove that the system comprised of $n$ out-of-order cores $P_{\text{so}}$ composed with a cache-coherent memory $M_{\text{cache}}$ implements a system comprised of the same cores composed with an atomic memory $M_{\text{atomic}}$, *i.e.,* $P^1_{\text{so}} + \ldots + P^n_{\text{so}} + M_{\text{cache}} \sqsubseteq P^1_{\text{so}} + \ldots + P^n_{\text{so}} + M_{\text{atomic}}$.

Using Theorem 10, which states that $P_{\text{so}} \sqsubseteq_{\text{noSpec}} P_{\text{dec}}$, and repeatedly using the trace-refines relation for non-interacting compositions, we can prove that $P^1_{\text{so}} + \ldots + P^n_{\text{so}} \sqsubseteq_{\text{noSpec}} P^1_{\text{dec}} + \ldots + P^n_{\text{dec}}$ (as **noSpec** performs a unique mapping). Using the trace-refines relation for interacting compositions, we can prove that the system comprised of $n$ out-of-order cores $P_{\text{so}}$ composed with the atomic memory $M_{\text{atomic}}$ refines a system comprised of $n$ decoupled cores $P_{\text{dec}}$ composed with the atomic memory $M_{\text{atomic}}$, *i.e.,* $P^1_{\text{so}} + \ldots + P^n_{\text{so}} + M_{\text{atomic}} \sqsubseteq P^1_{\text{dec}} + \ldots + P^n_{\text{dec}} + M_{\text{atomic}}$.

Using Theorem 11, which states that $P_{\text{dec}} + M_{\text{wrap}} \sqsubseteq P_{\text{inst}}$, and repeatedly using the trace-refines relation for non-interacting compositions, we can prove that $(P^1_{\text{dec}} + M^1_{\text{wrap}}) + \ldots + (P^n_{\text{dec}} + M^n_{\text{wrap}}) \sqsubseteq P^1_{\text{inst}} + \ldots + P^n_{\text{inst}}$. Using the trace-refines relation for interacting compositions, we can prove that $(P^1_{\text{dec}} + M^1_{\text{wrap}}) \ldots + (P^n_{\text{dec}} + M^n_{\text{wrap}}) +$

$M_{\text{atomic}} \sqsubseteq P^1_{\text{inst}} + \ldots + P^n_{\text{inst}} + M_{\text{atomic}}$. By rearranging the terms of this composition using the properties of commutativity and associativity (Theorem 5), we can derive that $P^1_{\text{dec}} + \ldots P^n_{\text{dec}} + M_{\text{atomic}} \sqsubseteq SC$.

Finally, using the transitivity property of trace-refines relation (Theorem 3), we can prove the final theorem, namely that a system composed of $n$ out-of-order cores connected to a cache-coherent memory implements the sequential consistency specification, *viz.*, $P^1_{\text{so}} + \ldots + P^n_{\text{so}} + M_{\text{cache}} \sqsubseteq SC$.

## 7.1 Removing Speculative Loads in the Labels of Atomic Memory

We will first prove that $M_{\text{atomic}} \sqsubseteq_{\text{noSpec}} M_{\text{atomic}}$ in this section. The proof is straightforward, but we spell it out in detail here as this serves as a high-level template for the other proofs.

**Theorem 9.** $M_{\text{atomic}} \sqsubseteq_{\text{noSpec}} M_{\text{atomic}}$

*Proof.* Since the module on both sides of the refinement is $M_{\text{atomic}}$, in order to avoid ambiguity we will use implementation to denote the module on the left of the refinement and specification to denote the one on the right of the refinement.

In general, we do the following in order to use Theorem 8.

1. *Providing the state map and rule map*: We provide a state map $\theta$ from an implementation state to a specification state and a *rule map* which, given a state $o$, maps a unit rule step of the implementation starting from state $o$ to a unit rule step in the specification. Just defining the state map and the rule map implicitly defines a step map $T$ since the label map is given in the trace-refines relation itself.

2. *Performing the compatibility check*: Whenever a unit step in the implementation is compatible with a unit step in the specification w.r.t. $T$, and the starting state of the specification is compatible with the starting state of the implementation w.r.t. $\theta$, then we prove that the ending state of the specification is also compatible with the ending state of the implementation w.r.t. $\theta$, and the label produced by the specification is compatible with the label produced by the implementation w.r.t. noSpec.

3. Proving that the specification is as combinable as the implementation w.r.t. $T$.

The state map $\theta$ between the implementation and the specification is obtained by removing all the entries in *rqFromC* and *toC* buffers that contain speculative loads, and rearranging the rest of the entries in the same order. The state in the monolithic memory is mapped to itself.

The rule map is as follows. There are two rules in $M_{\text{atomic}}$, **Rule**$\langle$*LdMem*$\rangle$ and **Rule**$\langle$*StMem*$\rangle$. We will also prove that the states after the compatible rule steps are compatible. Since $M_{\text{atomic}}$ does not call any external methods, no labels are produced during the steps. We will also perform the compatibility check when defining the rule map, for unit rule steps.

- **Rule**$\langle$*LdMem*$\rangle$: This can fire only when *rqFromC* has a first entry of the form $\langle \mathsf{Ld}, a, t \rangle$. If $t \neq \epsilon$ in the implementation, then the corresponding entry does not exist in the specification and we map it to a empty-rule step in the specification. The memory is unchanged and only a speculative load entry is removed, keeping the state obtained by applying $\theta$ on the changed state as before. This satisfies the compatibility check. If $t = \epsilon$, then the rule is mapped to its counterpart in the specification. The memory is again unchanged and the same return value is enqueued into *toC*, ensuring the compatibility check.

- **Rule**$\langle$*StMem*$\rangle$: This can fire only when *rqFromC* has a first entry of the form

$\langle \mathsf{St}, a, v\rangle$. This rule is mapped to its counterpart in the specification. The memory is changed identically in both the implementation and the specification, and the same return value is enqueued into $toC$, ensuring the compatibility check.

There are two methods defined in $M_{\text{atomic}}$, $rqFromC.enq$ and $toC.pop$. We must perform the compatibility checks for them. The labels of the unit steps executed by these methods are as follows:

- $\langle rqFromC.enq, \langle op, a, vt\rangle, ()\rangle$: If $op = \mathsf{Ld}$ and $vt \neq \epsilon$, *i.e.*, it is an enqueue of a speculative load, then noSpec removes this method, replacing it with $\epsilon$. So, the label map of this method in the specification is $\epsilon$, so this unit step is mapped to an empty-method step. Moreover, since the only state change happening when $op = \mathsf{Ld}$ and $vt \neq \epsilon$ is adding a speculative load request into $rqFromC$ buffer, $\theta$ applied on the changed state remains the same, satisfying the compatibility check. If $vt = \epsilon$ or $op \neq \mathsf{Ld}$, then the method is not removed by noSpec, and the same method step takes place in the specification, again satisfying the compatibility check.

- $\langle toC.pop, (), \langle op, \langle v, t\rangle\rangle\rangle$: As in the case of $rqFromC.enq$, if $op = \mathsf{Ld}$ and $t \neq \epsilon$, this step is replaced by an empty-method step; and if $t = \epsilon$ or $op \neq \mathsf{Ld}$, the same step takes place in the specification. This satisfies the compatibility check in the same manner as the unit method step producing $rqFromC.enq$.

Now we will prove that the specification is as combinable as the implementation w.r.t. unit step map $T$.

Whenever **Rule**$\langle LdMem\rangle$ is combined with method label $\langle rqFromC.enq, \langle \mathsf{Ld}, a, t\rangle, ()\rangle$, if $t \neq \epsilon$, then the method step is mapped to an empty-method step with which the rule mapping of **Rule**$\langle LdMem\rangle$ can be combined in the specification. Similarly, if the first element of $rqFromC$ buffer is $\langle \mathsf{Ld}, a, t\rangle$ where $t \neq \epsilon$, the rule mapping of **Rule**$\langle LdMem\rangle$

is an empty-rule step which can be combined with the $\langle rqFromC.enq, \langle op, a, vt\rangle, ()\rangle$ method step. Finally, if the method is $\langle rqFromC.enq, \langle op, a, vt\rangle, ()\rangle$ where $vt = \epsilon$ or $op \neq \mathsf{Ld}$ and the first element of *rqFromC* buffer is $\langle \mathsf{Ld}, a', vt'\rangle$ where $vt' = \epsilon$, then since the two steps do not write to the same register or call the same method (as the only shared state between the two steps resides inside the *rqFromC* buffer), the steps in the specification compatible with the corresponding steps in the implementation w.r.t. $T$ also do not write to the same register or call the same method and hence can also be combined.

Similarly, we can prove that when **Rule**$\langle LdMem\rangle$ is combined with method $toC.pop = \langle op, \langle v, t\rangle\rangle$, or when **Rule**$\langle StMem\rangle$ is combined with either defined method in the implementation, the steps in the specification compatible with the corresponding steps in the implementation w.r.t. $T$ can also be combined.

**Rule**$\langle LdMem\rangle$ and **Rule**$\langle StMem\rangle$ cannot be combined in the implementation because at least one of the combining steps must be a method step.

Finally, method $rqFromC.enq(op, a, vt)$ and $toC.pop$ do not contain any shared state or shared method calls both in the implementation and in the unit mapped steps in the specification. Thus they can be combined in the specification. □

## 7.2 Refinement Relation between Speculative Out-of-Order Processor and Instantaneous Processor

We will now focus our attention on proving refinement relations for the processor subsystem. As discussed earlier in the chapter, we will break the refinement relationship between the speculative out-of-order processor $P_{\text{so}}$ and the instantaneous processor $P_{\text{inst}}$ into two relationships, one between out-of-order processor and the decoupled processor $P_{\text{dec}}$ and the other between the decoupled processor and the instantaneous processor. We will first prove that the out-of-order processor implements the decou-

pled processor in Section 7.2.1, and then we will prove that the decoupled processor implements the instantaneous processor in Section 7.2.2.

### 7.2.1 Speculative Out-of-Order Processor Implements Decoupled Processor

In this section we will prove that a speculative out-of-order core $P_{\text{so}}$ implements a decoupled processor core $P_{\text{dec}}$. We will use the specification of the reorder buffer given in Invariant 1 in our proof.

**Theorem 10.** $P_{\text{so}} \sqsubseteq_{\text{noSpec}} P_{\text{dec}}$

*Proof.* Because of Theorem 8, the proof of this theorem is quite straightforward. There are no defined methods in $P_{\text{so}}$. So, we just have to provide a state map from a $P_{\text{so}}$ state to a $P_{\text{dec}}$ state and a rule map from every unit rule step of $P_{\text{so}}$ to a unit rule step of $P_{\text{dec}}$ such that they satisfy the compatibility check.

The state map from a state of $P_{\text{so}}$ to a state of $P_{\text{dec}}$ is given by dropping all the speculative state associated with $P_{\text{so}}$ and mapping the registers associated with the non-speculative state ($pc$, s and $w$) to their counterparts in the decoupled processor.

The only unit rule steps of $P_{\text{so}}$ that affect the non-speculative state are **Rule**⟨*NonMemory*⟩, **Rule**⟨*Halt*⟩, **Rule**⟨*LoadRq*⟩, **Rule**⟨*LoadRs*⟩, **Rule**⟨*LoadRsBad*⟩, **Rule**⟨*StoreRq*⟩ and **Rule**⟨*StoreRs*⟩. All these steps (except **Rule**⟨*LoadRsBad*⟩) are mapped to their counterparts in $P_{\text{dec}}$ and **Rule**⟨*LoadRsBad*⟩ is also mapped to **Rule**⟨*LoadRs*⟩. These steps use the value returned by the *rob.oldest* method of the reorder buffer in order to update the non-speculative state. Because of the constraint on the reorder buffer's correctness (Invariant 1), the value returned by *rob.oldest* is exactly what would have been returned by the *dec* and *exec* functions applied on the non-speculative state of $P_{\text{so}}$ before the step (and the value returned in the load response in the case of **Rule**⟨*LoadRs*⟩ or **Rule**⟨*LoadRsBad*⟩). Therefore, the state changes that these steps

make on the non-speculative state are the same as that made by their counterpart steps in $P_{\text{dec}}$, satisfying the compatibility check. All the other "speculative" unit rule steps in $P_{\text{so}}$ is mapped to empty-rule steps in $P_{\text{dec}}$, which also satisfy the compatibility check, since the speculative unit rule steps do not affect the state obtained after applying the state map. □

### 7.2.2 Decoupled Processor Implements Instantaneous Processor

We are finally left with $P_{\text{dec}} + M_{\text{wrap}} \sqsubseteq P_{\text{inst}}$ which we will prove in this section.

**Theorem 11.** $P_{\text{dec}} + M_{\text{wrap}} \sqsubseteq P_{\text{inst}}$

*Proof.* We will again use Theorem 8 to prove this theorem. The combined system $(P_{\text{dec}} + M_{\text{wrap}})$ does not have any defined methods after the composition. So, we just have to provide a state map from a $P_{\text{dec}} + M_{\text{wrap}}$ state to a $P_{\text{inst}}$ state and a rule map from every unit rule step of $P_{\text{dec}} + M_{\text{wrap}}$ to a unit rule step of $P_{\text{inst}}$.

The state map from a state of $(P_{\text{dec}} + M_{\text{wrap}})$ to a state of $P_{\text{inst}}$ is given by the concept of *flushing transitions* [59]. We apply the state changes enforced by the execution of **Rule**⟨*LoadRs*⟩ or **Rule**⟨*StoreRs*⟩ (depending on whether there is a load response or a store response). Effectively, a response in the *toC* buffer is drained and the registers of the processor are updated as if the flushing transition took place. The state-map function is given by projecting just the values of *pc* and *s* registers in the resulting state.

The mapping for the rules steps of $(P_{\text{dec}} + M_{\text{wrap}})$ are given as follows:

- **Rule**⟨*NonMem*⟩, **Rule**⟨*Halt*⟩: These are mapped to their counterparts in $P_{\text{inst}}$.
- **Rule**⟨*LoadRq*⟩, **Rule**⟨*StoreRq*⟩, **Rule**⟨*LoadRs*⟩ and **Rule**⟨*StoreRs*⟩: These are mapped to empty-rule steps in $P_{\text{inst}}$.

- **Rule**⟨*LdMem*⟩: This is mapped to **Rule**⟨*Load*⟩ in $P_{\text{inst}}$.
- **Rule**⟨*StMem*⟩: This is mapped to **Rule**⟨*Store*⟩ in $P_{\text{inst}}$.

In order to prove this theorem, we use Invariant 2 which says that the processor's $w$ state is set if and only if there is either a request from or a response to the processor, Invariant 3 which says that at most one request or one response can exist in $(P_{\text{dec}} + M_{\text{wrap}})$, and Invariant 4, Invariant 5, Invariant 6 and Invariant 7 which say that the request and response type matches that returned by executing the decode function *dec* on the current state. All these invariants are formally defined and proved below.

When **Rule**⟨*NonMem*⟩ or **Rule**⟨*Halt*⟩ execute, the $w$ state is not set, so there are no responses because of Invariant 2. So, the mapped state in $P_{\text{inst}}$ can be given by just reading the $s$ and $pc$ state in $(P_{\text{dec}} + M_{\text{wrap}})$ which is changed in the same manner in both $P_{\text{dec}}$ and $P_{\text{inst}}$.

When **Rule**⟨*LoadRq*⟩ or **Rule**⟨*StoreRq*⟩ execute, the $s$ and $pc$ registers remain the same after the step and again there are no responses because $w$ state is not set at the beginning of the transition. Since these rules are mapped to empty-rule steps, it is easy to see that the compatibility check holds.

When **Rule**⟨*LoadRs*⟩ or **Rule**⟨*StoreRs*⟩ execute, the $s$ and $pc$ registers are updated according to the respective steps. However, the state map function dictates that the update is already performed whenever a response is present to obtain the mapped state in $P_{\text{inst}}$. Moreover, Invariant 3 ensures that there can be at most one response (and hence only one flushing transition execution), and Invariant 6 and Invariant 7 ensure that these responses correspond to the current state of $s$ and $pc$ registers. Thus, the state map applied on the state of $P_{\text{dec}}$ at the beginning of the step is already the updated value because of a load or store response. Therefore the state map applied on the state of $P_{\text{dec}}$ does not change when either of these two unit rule steps take place. Since these unit rule steps are mapped to empty-rule steps, the compatibility check holds.

When **Rule**$\langle$*LdMem*$\rangle$ executes, method *ldRq* is called. The rule in the specification given by the rule map for **Rule**$\langle$*LdMem*$\rangle$ is **Rule**$\langle$*Load*$\rangle$, which also calls *ldRq* method. Because of Invariant 4, the arguments with which these methods are called in the two modules are the same. At the end of the unit rule step, an appropriate response is enqueued into the *toC* buffer in $P_{\text{dec}}$, while the state is changed in $P_{\text{inst}}$. According to the state map given by the flushing transition, and because there can only be one response (Invariant 3), the compatibility check holds. A similar argument holds when **Rule**$\langle$*StMem*$\rangle$ executes, which is mapped to **Rule**$\langle$*Store*$\rangle$. □

The decoupled core ($P_{\text{dec}} + M_{\text{wrap}}$) obeys the following two invariants which are used in the proof that it implements the instantaneous processor $P_{\text{inst}}$.

**Invariant 2.** The wait state $w$ is set iff there is a request in the *rqFromC* buffer or a response in the *toC* buffer.

**Invariant 3.** The total number of requests in the *rqFromC* buffer and responses in the *toC* buffer cannot exceed 1.

**Invariant 4.** If the *rqFromC* buffer has a $\langle \mathsf{Ld}, a, t\rangle$ request, then $dec(s, pc, getInst(pc)) = \langle \mathsf{Ld}, a, _\rangle$ and $t = \epsilon$.

**Invariant 5.** If the *rqFromC* buffer has a $\langle \mathsf{St}, a, v\rangle$ request, then $dec(s, pc, getInst(pc)) = \langle \mathsf{St}, a, v\rangle$.

**Invariant 6.** If the *toC* buffer has a $\langle \mathsf{Ld}, v\rangle$ response, then $dec(s, pc, getInst(pc)) = \langle \mathsf{Ld}, _, _\rangle$.

**Invariant 7.** If the *toC* buffer has a $\langle \mathsf{St}\rangle$ response, then $dec(s, pc, getInst(pc)) = \langle \mathsf{St}, _, _\rangle$.

**Lemma 3.** Invariant 2 to Invariant 7 hold after zero or more steps of $P_{\text{dec}}$.

*Proof.* When **Rule**⟨*NonMem*⟩ or **Rule**⟨*Halt*⟩ fire, the $w$ state remains not set, so there are no requests or responses, and the steps do not add any request or responses, satisfying all the invariants.

When **Rule**⟨*LoadRq*⟩ or **Rule**⟨*StoreRq*⟩ fire, the $w$ state changes from False to True, which means there are no requests or responses at the beginning of the step. After the step, an appropriate request is added while there are still no responses. This satisfies all the invariants.

When **Rule**⟨*LoadRs*⟩ or **Rule**⟨*StoreRs*⟩ fire, the $w$ state changes from True to False, so there is exactly one appropriate response (and no requests) at the beginning of the step. After the step, the response is removed and no request is added, thus satisfying all the invariants.

When **Rule**⟨*LdMem*⟩ or **Rule**⟨*StMem*⟩ fire, since there is a request, the $w$ state is set at the beginning of the step, and remains unchanged. The request is removed and an appropriate response is added, thus satisfying all the invariants. □

## 7.3 Conclusion

In this chapter we decomposed the proof of correctness of the complex multiprocessor system into multiple sub-proofs using the properties mentioned in Section 6.5. We prove one of the sub-goals, *viz.*, that the complex processor refines an instantaneous processor, in this chapter. In the next chapter we will prove that the cache hierarchy is coherent, *i.e.*, it implements the atomic memory specification. Once we have a proof for that, we have shown in this chapter that the sub-proofs can be combined to verify the full system.

# Chapter 8

# Verifying the Memory Subsystem

In this chapter we prove that the memory subsystem consisting of the hierarchy of coherent caches implements the atomic memory specification. This constitutes a soundness condition, *i.e.*, it shows that the behaviors exhibited by the cache hierarchy are sound with respect to those exhibited by the atomic memory. In addition to that, we will prove that the cache hierarchy is deadlock-free, *i.e.*, there is always at least one (state-changing) step that can take place if there are pending requests from processors, and is livelock-free, *i.e.*, any pending request from processors will eventually be serviced.

We will be formalizing these notions of soundness, deadlock-freedom and livelock-freedom and prove these properties in Sections 8.1, 8.2 and 8.3, respectively.

## 8.1 Soundness of a Hierarchy of Coherent Caches w.r.t. Atomic Memory

In this section, we prove that the hierarchy of coherent caches $M_{\text{cache}}$ implements the atomic memory specification $M_{\text{atomic}}$.

**Theorem 12.** $M_{\text{cache}} \sqsubseteq M_{\text{atomic}}$

*Proof.* In order to prove this theorem, we will first decompose the $M_{\text{cache}}$ and $M_{\text{atomic}}$ modules. As seen in Definition 6, the $M_{\text{cache}}$ module is made up of L1 caches, $\text{L}_{\text{int}}$ caches and the $M$ module. We will strip out the *cRqs* and *toC* buffers from every L1 cache and call the resulting module $M'_{\text{cache}}$. Because of commutativity and associativity of refinements (Theorem 5), we have

$$M_{\text{cache}} \equiv M'_{\text{cache}} + (cRqs^1 + \ldots + cRqs^n) + (toC^1 + \ldots + toC^n)$$

where $cRqs^1, \ldots, cRqs^n$ and $toC^1, \ldots, toC^n$ denote the buffers in the L1 caches which are accessed by the processors.

Similarly, we can also strip out *cRqs* and *toC* buffers from the $M_{\text{atomic}}$ module to get $M'_{\text{atomic}}$. Using Definition 2 and Definition 3, and applying commutativity and associativity of refinements again, we get

$$M'_{\text{atomic}} \equiv M'^{1}_{\text{wrap}} + \ldots M'^{n}_{\text{wrap}} + M_{\text{inst}}$$
$$M_{\text{atomic}} \equiv M'_{\text{atomic}} + (cRqs^1 + \ldots + cRqs^n) + (toC^1 + \ldots + toC^n)$$

Since we are using the same names for the *cRqs* and *toC* buffers in both $M_{\text{cache}}$ and $M_{\text{atomic}}$, in order to avoid ambiguity in this proof, we will use the subscripts **cache** and **atomic** to denote the buffers belonging to $M_{\text{cache}}$ and $M_{\text{atomic}}$, respectively.

The $toC_{\text{cache}}$ buffers are exactly the same as the $toC_{\text{atomic}}$ buffer, and thus $toC_{\text{cache}} \sqsubseteq toC_{\text{atomic}}$ because of reflexivity (Theorem 2). We will be proving in Theorem 13 below that $cRqs_{\text{cache}} \sqsubseteq_{\mathsf{noUpd}} cRqs_{\text{atomic}}$, where **noUpd** removes the *get*, *upd* and *all* methods from the labels. Using these properties and applying the non-interacting composition theorem (Theorem 6) repeatedly, we can prove that $(cRqs^1_{\text{cache}} + \ldots + cRqs^n_{\text{cache}}) + (toC^1_{\text{cache}} + \ldots + toC^n_{\text{cache}}) \sqsubseteq_{\mathsf{noUpd}} (cRqs^1_{\text{atomic}} + \ldots_{\text{atomic}} + cRqs^n_{\text{atomic}}) + (toC^1_{\text{atomic}} + \ldots + toC^n_{\text{atomic}})$ (as **noUpd** satisfies **uniqMap** predicate). We will finally prove in Theorem 14 that $M'_{\text{cache}} \sqsubseteq_{\mathsf{noUpd}} M'_{\text{atomic}}$. With these results, using the interacting composition the-

orem (Theorem 7), we can prove that $M_{\mathsf{cache}}$ implements $M_{\mathsf{atomic}}$. ☐

Proving that $cRqs_{\mathsf{cache}} \sqsubseteq_{\mathsf{noUpd}} cRqs_{\mathsf{atomic}}$ is straightforward as can be seen below. We first define the mapping function noUpd.

**Definition 40.**

$$\mathsf{noUpd}\ f \triangleq \begin{cases} \langle x, ()\rangle : f = \langle cRqs_{\mathsf{cache}}.ins, x, ()\rangle \\ \langle n, v\rangle : f = \langle cRqs_{\mathsf{cache}}.extract, n, v\rangle \\ \epsilon : \text{otherwise} \end{cases}$$

**Theorem 13.** $cRqs_{\mathsf{cache}} \sqsubseteq_{\mathsf{noUpd}} cRqs_{\mathsf{atomic}}$.

*Proof.* First note that the only difference between the states in the two modules is that $cRqs_{\mathsf{cache}}$ stores an augmented state Init, ⟨WaitSt, _⟩ or ⟨WaitV, _⟩ for each valid entry in the buffer in addition to the values stored in $cRqs_{\mathsf{atomic}}$. Module $cRqs_{\mathsf{cache}}$ also defines some additional methods compared to $cRqs_{\mathsf{atomic}}$. Of these methods, *get*, *read* and *all* do not change the state of $cRqs_{\mathsf{cache}}$, while *upd* only affects the augmented state of the module. Thus using Theorem 8 to prove this theorem is straightforward; the state map is obtained by ignoring the augmented state in each buffer entry and mapping the rest of the message in each entry to itself; and there are no rules in $cRqs_{\mathsf{cache}}$ so there is no need for a rule map. ☐

Proving that $M'_{\mathsf{cache}} \sqsubseteq M'_{\mathsf{atomic}}$ is the goal of our next theorem.

**Theorem 14.** $M'_{\mathsf{cache}} \sqsubseteq_{\mathsf{noUpd}} M'_{\mathsf{atomic}}$.

*Proof.* We prove this lemma from first principles by induction on a multistep of $M'_{\mathsf{cache}}$ that constitutes its behavior. We show that whenever $M'_{\mathsf{cache}}$ produces a sequence of labels, then $M'_{\mathsf{atomic}}$ will produce a sequence of labels which is compatible with the former sequence w.r.t. noUpd.

Initially, no labels are present in both $M'_{\mathsf{cache}}$ and $M'_{\mathsf{atomic}}$.

Let us assume that the induction hypothesis holds at the current state. So the sequence of labels generated by $M_{\text{atomic}}$ is compatible with the sequence of labels generated by $M'_{\text{cache}}$ w.r.t. noUpd. We will prove that for any step of $M'_{\text{cache}}$ executed next, there exists a step of $M'_{\text{atomic}}$ that can produce a label compatible with that produced by the step of $M'_{\text{cache}}$ w.r.t. noUpd.

Whenever a step of $M'_{\text{cache}}$ is executed by **Rule**⟨*LdHit*⟩ or **Rule**⟨*LdDeferred*⟩ in the $i^{\text{th}}$ L1 cache, we will have a step of $M'_{\text{atomic}}$ executed by **Rule**⟨$LdMem^i_n$⟩ in where $n$ is the *cRqs* entry which is extracted in the step of $M'_{\text{cache}}$. Similarly, whenever a step of $M'_{\text{cache}}$ is executed by **Rule**⟨*StHit*⟩ or **Rule**⟨*StDeferred*⟩ in the $i^{\text{th}}$ L1 cache of $M'_{\text{cache}}$, we will have a step of $M'_{\text{atomic}}$ executed by **Rule**⟨$StMem^i_n$⟩ where $n$ is again the *cRqs* entry which is extracted in the step of $M'_{\text{cache}}$. We will map every other step of $M'_{\text{cache}}$ to empty-rule steps of $M'_{\text{atomic}}$. We now have to prove that whenever a step of $M'_{\text{cache}}$ takes place, when the corresponding step of $M'_{\text{atomic}}$ takes place, the label that the latter step produces is compatible with the label produced by the former step w.r.t. noUpd. We will call this the *label-compatibility check* for a step of $M'_{\text{cache}}$ in the rest of this proof.

It is easy to see that for all the steps of $M'_{\text{cache}}$ which are executed by rules other than **Rule**⟨*LdHit*⟩ and **Rule**⟨*LdDeferred*⟩, the label-compatibility check holds.

Proving that the label-compatibility check holds for steps executed by **Rule**⟨$LdHit^i$⟩ and **Rule**⟨$LdDeferred^i$⟩ is very difficult. This boils down to proving that whenever a step of $M'_{\text{cache}}$ produces a label, which on application of noUpd produces the called-methods set $\{\langle cRqs^i.extract, n, \langle \mathsf{Ld}, va, t\rangle, toC^i.enq, \langle \mathsf{Ld}, data[l][getOffset(va)]\rangle\rangle\}$ matches the label produced by the corresponding step of $M'_{\text{atomic}}$ executed by **Rule**⟨$LdMem^i_n$⟩, *viz.*, the called-methods set $\{\langle cRqs^i.extract, n, \langle \mathsf{Ld}, va, t\rangle\rangle, \langle toC^i.enq, \langle \mathsf{Ld}, M_{\text{inst}}.ldRq^i(va)\rangle\rangle\}$. Thus, we have to prove that $data[l][getOffset(va)]$ matches $M_{\text{inst}}.ldRq^i(va)$.

In order to prove this, we will first prove that the value in $data[l][getOffset(va)]$ and that returned by $M_{\text{inst}}.ldRq^i_n(va)$ are both the *latest values* for address $va$. We

will formally define "latest value" below. Before that, for notational convenience, we will define what we mean by a store for address $va$, namely $store(va, v)$.

**Definition 41.** A step is said to be a store for address $va$, *i.e.,* $store(va, v)$, iff it produces labels on which if the noUpd function is applied, it becomes $\langle cRqs.extract, _, \langle \mathsf{St}, va, v\rangle\rangle$ and $\langle toC.enq, \langle St\rangle, ()\rangle$.

From this, we can define latest value as below:

**Definition 42.** $latestValue(\text{va})$ in the current state is either

- $m_0(va)$, *i.e.,* the initial value of word address $va$ if $store(va, _)$ has not yet taken place, or
- $v$, if $store(va, v)$ has taken place and no $store(va, _)$ has taken place after that.

Proving that $M_{\text{inst}}.ldRq_n^i(va) = latestValue(va)$ is straightforward. From Invariant 10 below we get that for every word address $va$, $mem[va]$ in $M'_{\text{atomic}}$ contains the latest value. Since $M_{\text{inst}}.ldRq_n^i(va)$ returns $mem[va]$, it gives the latest value for word address $va$.

Proving that $data[l][getOffset(va)] = latestValue(va)$ is more involved. We will be using Invariant 8 which states that whenever a line address is *clean* in a cache, then all the words in the line contain the latest value. For this, we need to define what "clean" is. In order to define that, we will first overload the names of the register arrays (like *cs*, *dir*, *etc.*) to denote mappings from line addresses to values. We will be using the variable $a$ to denote addresses and $l$ to denote slots; this will remove any ambiguity between slot-to-value mapping (*i.e.,* simple register array accesses) and address-to-value mapping. To define such a mapping, we need to define a function $toSlot(a)$, which maps a line address $a$ to a slot in the cache or $\epsilon$ if that address is not found in the cache. In case of the memory $M'$, $toSlot(a) = a$ since the memory contains all the addresses. For all other caches, *toSlot* is defined as shown below. It

uses two functions *searchTags* and *searchAddrWaitSt* in its definition. The function $searchTags(cs, tags, a)$, as defined earlier (Definition 4-3c), returns the slot containing line address $a$ in a non-$I$ state (or $\epsilon$ if no such slot is present) in the L1 and internal caches. Function *searchAddrWaitSt* is defined below.

**Definition 43.**

$$searchAddrWaitSt(cRqs.all, a) \triangleq$$

$$\begin{cases} l : \text{current cache is L1 and } \langle\langle \mathsf{WaitSt}, l\rangle, \langle op, va, vt\rangle\rangle \in cRqs.all \wedge getTag(va) = a \\ a : \text{current cache is memory and } \langle \mathsf{WaitSt}, \langle c, a, y, x\rangle\rangle \in cRqs.all \\ l : \text{current cache is an internal cache and } \langle\langle \mathsf{WaitSt}, l\rangle, \langle c, a, y, x\rangle\rangle \in cRqs.all \\ \epsilon : \text{otherwise} \end{cases}$$

**Definition 44.**

$$toSlot(a) \triangleq \begin{cases} searchTags(cs, tags, a) : searchTags(cs, tags, a) \neq \epsilon \\ searchAddr(cRqs.all, a) : searchAddrWaitSt(cRqs.all, a) \neq \epsilon \\ \epsilon : \text{otherwise} \end{cases}$$

Once *toSlot* is defined, then the value corresponding to address $a$ is given by the position $toSlot(a)$ in the corresponding register array if $toSlot(a) \neq \epsilon$, or by the default value if $toSlot(a) = \epsilon$.

**Definition 45.**

$$cs[a] \triangleq \begin{cases} cs[toSlot(a)] : toSlot(a) \neq \epsilon \\ I : toSlot(a) = \epsilon \end{cases}$$

$$w[a] \triangleq \begin{cases} w[toSlot(a)] : toSlot(a) \neq \epsilon \\ \textsf{False} : toSlot(a) = \epsilon \end{cases}$$

$$dir[a][c] \triangleq \begin{cases} dir[toSlot(a)][c] : toSlot(a) \neq \epsilon \\ I : toSlot(a) = \epsilon \end{cases}$$

$$dirw[a][c] \triangleq \begin{cases} dirw[toSlot(a)][c] : toSlot(a) \neq \epsilon \\ \textsf{False} : toSlot(a) = \epsilon \end{cases}$$

$$data[a] \triangleq \begin{cases} data[toSlot(a)] : toSlot(a) \neq \epsilon \\ 0 : toSlot(a) = \epsilon \end{cases}$$

Now we can define whether an address $a$ is *clean* in a cache as follows:

**Definition 46.** An address $a$ is said to be *clean* in a cache iff the cache state for that address is at least $S$ and the directory states for all its children (if they exist) for that address are at most $S$.

$$clean(a) \triangleq cs[a] \geq S \wedge (\forall c \in children.\ dir[a][c] \leq S)$$

Coming back to our main theorem, if we can prove that $clean(a)$ is true (where $a = getTag(va)$), in the current L1 cache performing the step, then $data[a]$ will contain the latest value by Invariant 8, which we state below; we will prove it later.

**Invariant 8.** If an address $a$ in a cache is *clean*, then the data in that address is the latest value for that address.

$$\forall a.\ clean(a) \Rightarrow \forall o.\ data[a][o] = latestValue(a \mathbin{++} o)$$

where $a \mathbin{++} o$ denotes the word address created from the line address $a$ and the offset $o$.

In particular, $data[a][getOffset(va)]$ will also contain the latest value. If we prove that $toSlot(a) = l$, it completes the theorem, since in **Rule**$\langle LdHit \rangle$ and **Rule**$\langle LdDeferred \rangle$, $cs[l] \geq S$ and L1 caches have no children thereby making $clean(a)$ true.

To prove that $toSlot(a) = l$ in **Rule**$\langle LdHit \rangle$ and **Rule**$\langle LdDeferred \rangle$, we use Invariant 9 which we state below and prove later.

**Invariant 9.** Whenever $searchTags(cs, tags, a) \neq \epsilon$ or $searchAddrWaitSt(cRqs.all, a) \neq \epsilon$ then $toSlot(a)$ is given by the slot returned by either.

$$\forall l \neq \epsilon.\ \wedge (searchTags(cs, tags, a) = l \vee searchAddrWaitSt(cRqs.all, a) = l) \Rightarrow toSlot(a) = l$$

In **Rule**$\langle LdHit \rangle$, $searchTags(cs, tags, a) = l$ and in **Rule**$\langle LdDeferred \rangle$, $cRqs.read(n)$ returns $\langle \langle \mathsf{WaitSt}, l \rangle, \langle \mathsf{Ld}, va, t \rangle \rangle$ where $getTag(va) = a$. □

We now state and prove the latest-value invariant for the instantaneous memory $M_{\text{inst}}$ which is a component of $M'_{\text{atomic}}$.

**Invariant 10.** The memory *mem* in $M_{\text{inst}}$ always contains the latest value for every address.

$$\forall va.\ mem[va] = latestValue(va)$$

**Lemma 4.** Invariant 10 holds after zero or more steps of $M'_{\text{atomic}}$.

*Proof.* Initially, this invariant trivially holds because *mem* is initialized with the initial value of memory $m_0$.

Let us assume that this invariant holds in the current state. We have to prove that any new step will preserve the invariant. There are two steps to consider: **Rule**⟨*LdMem*⟩ and **Rule**⟨*StMem*⟩.

In **Rule**⟨*LdMem*⟩, the register array *mem* remains unchanged. So, the invariant holds.

In **Rule**⟨*StMem*⟩, $mem[va]$ is updated to $v$, which is supplied through the label $stRq(va, v)$, again preserving the invariant. □

In order to prove Invariant 9 and Invariant 8, we need to prove several other invariants. We will stating these invariants and some guidelines on how to prove them in the next section.

### 8.1.1 Cache Invariants

Before we begin with the proof, we will be defining the notation that we will be using throughout this section.

### 8.1.2 Notation used in the Invariants and the Proofs

We call a request in the *cRqs* buffer in the ⟨WaitSt, _⟩ or ⟨WaitV, _⟩ augmented state a *waiting request*. We also use Init-, $\langle \mathsf{WaitSt}, l\rangle$- or $\langle \mathsf{WaitV}, l\rangle$-request for a request in *cRqs* buffer in the Init, $\langle \mathsf{WaitSt}, l\rangle$ or $\langle \mathsf{WaitV}, l\rangle$ augmented state, respectively. We say that a slot $l$ is in wait state whenever $w[l]$ is set, and address $a$ is in wait state whenever $w[a]$ is set.

Sometimes it is useful to talk about the "next state" after a step has happened. We use a function $\mathsf{next}(e)$ to denote the value of expression $e$ at the end of a step, as opposed to just $e$, which denotes the value of $e$ at the beginning of the step.

We use $\mathit{cRqsRecv}(r)$ to denote that a request $r$ is extracted from *cRqs* in a step. Similarly, $\mathit{rsFromCRecv}(r)$, $\mathit{pRqsRecv}(r)$ and $\mathit{rsFromPRecv}(r)$ denote a response popped from the *rsFromC* buffer, a request extracted from the *pRqs* buffer or a response popped from the *fromP* buffer, respectively, during a step. In the same vein, we have $\mathit{rqToCSend}(r)$, $\mathit{rsToCSend}(r)$, $\mathit{rqToPSend}(r)$ and $\mathit{rsToPSend}(r)$ to denote a request enqueued into the *toC* buffer, a response enqueued into the *toC* buffer, a request enqueued into the *rqToP* buffer or a response enqueued into the *rsToP* buffer, respectively, during a step.

A message is said to be *sent* if it is enqueued into the appropriate buffer at the source cache, and is said to be *received* if it is extracted or popped, as the case may be, from the appropriate buffer at the destination cache.

We use *rqsFCTP* to denote the set of requests which are currently *in-flight* from a child to its parent $p$. By in-flight requests, we mean all the requests $r$ such that $\langle p, r\rangle$ is in the *rqToP* buffer of the cache or the network buffers, or $r$ is in the *rqFromC* buffer of the parent or a $\langle _, r\rangle$ entry is in *cRqs* buffer of the parent. Similarly, *rssFCTP* denotes the set of responses which are in-flight to parent $p$ of a cache, *i.e.,* responses $r$ such that $\langle p, r\rangle$ is in the *rsToP* buffer of the cache or the network buffers or $r$ is in the *rsFromC* buffer of the parent.

In the same manner as before, we use *rqsFPTC* to denote the set of requests in-flight from a parent to its children. This includes tuples of the form $\langle c, a, x\rangle$ such that $\langle c, \langle \mathsf{Req}, a, x\rangle\rangle$ is in the *toC* buffer of the parent, or in the network buffers, or $\langle \mathsf{Req}, a, x\rangle$ is in the *fromP* buffer of child $c$ or $\langle a, x\rangle$ is in the *pRqs* buffer of child $c$. Similarly, *rssFPTC* denotes the set of responses in-flight from a parent to its children. This includes responses of the form $\langle c, a, x\rangle$ such that $\langle c, \langle \mathsf{Rs}, a, x\rangle\rangle$ is in the *toC* buffer of the cache, or in the network buffers, or $\langle \mathsf{Rs}, a, x\rangle$ is in the *fromP* buffer of child $c$.

We use $\mathit{rqsNumFCTP}(a)$, $\mathit{rssNumFCTP}(a)$, $\mathit{rqsNumFPTC}(a)$ and $\mathit{rssNumFPTC}(a)$ to denote, from the perspective of a specific child (which should be obvious from the

context), the number of requests to the child's parent, responses to its parent, requests from its parent and responses from its parent, respectively, that are currently in-flight for address $a$.

We define projection functions to obtain various fields of a message. For a request message $r = \langle c, a, y, x\rangle$ from a child cache $c$ to its parent, we have $child(r) = c, addr(r) = a, fromState(r) = y$ and $toState(r) = x$. For a response message $r = \langle c, a, x, d\rangle$ from a child cache $c$ to its parent, we have $child(r) = c, addr(r) = a, toState(r) = x$ and $\mathsf{lineData}(r) = d$. For a request message $r = \langle a, x\rangle$ or $r = \langle \mathsf{Req}, a, x\rangle$ from a parent to its child $c$, we have $child(r) = c, addr(r) = a$ and $toState(r) = x$. And, for a response message $r = \langle \mathsf{Rs}, a, x, d\rangle$ from a parent to its child $c$, we have $child(r) = c$, $addr(r) = a$, $toState(r) = x$ and $\mathsf{lineData}(r) = d$. We essentially overload the names of the projection functions depending on the type of the message on which each operates.

For requests to L1 caches from the processors, we use the projection functions again, but we obtain line address instead of word address from these requests. More precisely, for a request $r = \langle op, va, vt\rangle$, we have $addr(r) = getTag(va), toState(r) =$ **if** $(op = \mathsf{Ld})$ **then** $S$ **else** $M$ and $\mathsf{wordData}(r) = vt$.

### 8.1.3 Formal Specification of Cache Invariants

We are now ready to state all the invariants. These invariants hold for all caches except for invariants involving the directory state *dir* or the directory-wait state *dirw*, which do not apply to L1 caches. For the purposes of these invariants, we will extend the notions of $cs, tags, w, toSlot, searchTags$ and $searchSlot$ for the memory module $M$ (Definition 8). In the memory, the address is synonymous with the slot. For any slot $l$ in the memory, $cs[l] = M, tags[l] = l, w[l] =$ False and $searchSlot(cRqs.all, l) = searchAddr(cRqs.all, l)$. For any address $a$ in the memory, $toSlot(a) = a$ and $searchTags(_, _, a) = a$.

### Proving Invariant 9

We begin with Invariant 9, which was used in the proof of Theorem 14. In order to prove this invariant, we need the following invariants. Once these invariants are proved, it is very easy to prove Invariant 9 and hence we omit its proof.

**Invariants Relating Tags of Slots and Requests in the *cRqs* buffer:** The following list of invariants relate the addresses in the tags of slots and those in the *cRqs* buffer.

**Invariant 11.** The tags in each non-$I$ slot are unique.

$$\forall l_1\ l_2.\ cs[l_1] \neq I \wedge cs[l_2] \neq I \wedge tags[l_1] = tags[l_2] \Rightarrow l_1 = l_2$$

**Invariant 12.** The slots associated with each waiting request in a *cRqs* buffer are unique.

$$\forall n_1\ n_2\ l.\ cRqs.read(n_1) = \langle\langle _, l\rangle, _\rangle \wedge cRqs.read(n_2) = \langle\langle _, l\rangle, _\rangle \Rightarrow n_1 = n_2$$

**Invariant 13.** The addresses in each waiting request in a *cRqs* buffer are unique.

$$\forall n_1\ n_2\ r_1\ r_2.\ cRqs.read(n_1) = \langle\langle _, _\rangle, r_1\rangle \wedge$$
$$cRqs.read(n_2) = \langle\langle _, _\rangle, r_2\rangle \wedge addr(r_1) = addr(r_2) \Rightarrow n_1 = n_2$$

**Invariant 14.** If there is a $\langle \mathsf{WaitSt}, l\rangle$-request in the *cRqs* buffer and if $cs[l] \neq I$, then slot $l$ contains the address associated with the request.

$$\forall n\ l.\ cRqs.read(n) = \langle\langle \mathsf{WaitSt}, l\rangle, r\rangle \wedge cs[l] \neq I \Rightarrow addr(r) = tags[l]$$

**Invariant 15.** If some non-$I$ slot $l$ contains the address of some waiting request in

the *cRqs* buffer, then it is a $\langle \mathsf{WaitSt}, l\rangle$-request.

$$\forall n\ l\ r.\ cs[l] \neq I \wedge cRqs.read(n) = \langle\langle _, _\rangle, r\rangle \wedge tags[l] = addr(r) \Rightarrow \\ cRqs.read(n) = \langle\langle \mathsf{WaitSt}, l\rangle, r\rangle$$

**Invariant 16.** Whenever a slot $l$ is in wait state, then there is a $\langle \mathsf{WaitSt}, l\rangle$ request in the *cRqs* buffer and the cache state of the slot $l$ is less than *toState* of the request.

$$\forall n\ l.\ w[l] \wedge cRqs.read(n) = \langle\langle \mathsf{WaitSt}, l\rangle, r\rangle \Rightarrow cs[l] < toState(r)$$

**Invariant 17.** If a slot $l$ is in wait state and there is a $\langle \mathsf{WaitSt}, l\rangle$-request associated with address $a$, then there is a *consistent* request for address $a$ to the parent from that cache, or a consistent response for address $a$ from the parent to that cache.

$$\forall n\ l\ r.\ w[l] \wedge cRqs.read(n) = \langle\langle \mathsf{WaitSt}, l\rangle, r\rangle \Rightarrow \\ \exists r'.\ addr(r') = addr(r) \wedge toState(r) = toState(r') \wedge \\ ((r' \in rqsFCTP \wedge cs[l] \leq fromState(r')) \vee r' \in rssFPTC)$$

**Invariant 18.** The sum of the number of requests in-flight to the parent and the number of responses in-flight from the parent associated with the same address cannot exceed one.

$$rqsNumFCTP(a) + rssNumFPTC(a) \leq 1$$

**Invariant 19.** If there is a request from a cache to its parent associated with address $a$, then in the cache, there is a *consistent* $\langle \mathsf{WaitSt}, l\rangle$-request associated with address

$a$ in the *cRqs* buffer, and slot $l$ is in wait state.

$$\forall a\ r.\ r \in \mathit{rqsFCTP} \Rightarrow$$
$$\exists n\ l\ r'.\ w[l] \wedge \mathit{cRqs.read}(n) = \langle\langle \mathsf{WaitSt}, l\rangle, r'\rangle \wedge \mathit{addr}(r') = \mathit{addr}(r) \wedge$$
$$\mathit{toState}(r') = \mathit{toState}(r) \wedge \mathit{cs}[l] \leq \mathit{fromState}(r) < \mathit{toState}(r)$$

**Invariant 20.** If there is a response from the parent to the cache associated with address $a$, then in the cache, there is a *consistent* $\langle \mathsf{WaitSt}, l\rangle$-request associated with address $a$ in the *cRqs* buffer, and slot $l$ is in wait state.

$$\forall a\ r.\ r \in \mathit{rssFPTC} \Rightarrow$$
$$\exists n\ l\ r'.\ w[l] \wedge \mathit{cRqs.read}(n) = \langle\langle \mathsf{WaitSt}, l\rangle, r'\rangle \wedge \mathit{addr}(r') = \mathit{addr}(r) \wedge$$
$$\mathit{toState}(r) = x \wedge \mathit{cs}[l] < \mathit{toState}(r)$$

All the above invariants (Invariant 11 to Invariant 20) can be proved simultaneously by induction on a multistep.

### Proving Invariant 8

In order to prove Invariant 8, we need several other invariants which are discussed below. We will start with invariants about the maintenance of order between messages sent between caches.

**Invariants on Ordering of Messages:** We first give several invariants related to the point-to-point ordering of request and response messages, *i.e.*, invariants putting constraints on when one message can overtake another sent from the same source to the same destination.

**Invariant 21.** Responses from a child to its parent are popped from the *rsFromC*

buffer at the parent in the order they were enqueued.

**Invariant 22.** Responses from a parent to a child are popped from the *fromP* buffer at the child in the order they were enqueued for the same address.

**Invariant 23.** Requests from a child to its parent are received at the parent in the order they were sent by the child for the same address.

**Invariant 24.** Any request in the *pRqs* buffer is enqueued by a child earlier than any message enqueued by the same child in the *fromP* buffer.

**Invariant 25.** A response from the parent cannot be popped by a child from the *fromP* buffer before a request enqueued by the parent earlier for the same address has been extracted from the *pRqs* buffer by the child.

**Invariant 26.** A request from the parent cannot be extracted by a child from the *pRqs* buffer before a response enqueued by the parent earlier for the same address has been extracted from *rsToP* buffer by the child.

All these invariants can be proved easily. Most of them use the ordering constraints of the network, Some require induction on the multistep; and Invariants 22 and 23 use Invariant 18 in their proofs.

**Invariants Using Address-to-Value Mappings:** While the invariants until now are properties about slot-to-value mappings, the rest of the invariants in this section pertain to address-to-value mappings. For proving those invariants, it is helpful to use the following invariant, which ensures that an address-to-value mapping for a particular address changes during a step only if the address is already mapped to a slot and the slot-to-value mapping for the mapped slot changes during that step. This is because this invariant says that whenever a new non-$\epsilon$ slot mapping is created for an address, the values corresponding to that slot are all at their defaults ($I$ or False).

So a step creating such a new slot mapping for an address cannot change some value mapped to that address from its default. Thus, this theorem prevents analyzing a lot of unnecessary Bluespec rules when proving the later invariants.

**Invariant 27.** Whenever $\mathit{toSlot}(a)$ changes from $\epsilon$ to $l \neq \epsilon$, then at the end of the step the registers in slot $l$ are at their default values.

$$\forall a\ l.\ \mathit{toSlot}(a) = \epsilon \wedge \mathsf{next}(\mathit{toSlot}(a)) = l \neq \epsilon \Rightarrow$$
$$\mathsf{next}(\mathit{cs}[l]) = I \wedge \mathsf{next}(w[l]) = \mathsf{False} \wedge$$
$$(\forall c.\ \mathsf{next}(\mathit{dir}[l][c]) = I \wedge \mathsf{next}(\mathit{dirw}[l][c]) = \mathsf{False})$$

This can be proven using Invariants 16 and 12. We also need to use the fact that whenever a directory-wait state is set, then the directory state cannot be $I$, which can be proved directly using induction on a multistep.

The proofs for the rest of the invariants all use Invariants 27 and 9 in their proofs.

**Invariant 28.** The cache state of an address in a cache is downgraded iff a response for that address is enqueued into the *rsToP* buffer, and it is upgraded iff a response for that address is popped from the *fromP* buffer.

$$\forall a.$$
$$(\mathit{cs}[a] > \mathsf{next}(\mathit{cs}[a]) \Leftrightarrow \exists r.\ \mathit{rsToPSend}(r) \wedge \mathit{addr}(r) = a \wedge \mathsf{next}(\mathit{cs}[a]) = \mathit{toState}(r)) \wedge$$
$$(\mathit{cs}[a] < \mathsf{next}(\mathit{cs}[a]) \Leftrightarrow \exists r.\ \mathit{rsFromPRecv}(r) \wedge \mathit{addr}(r) = a \wedge \mathsf{next}(\mathit{cs}[a]) = \mathit{toState}(r))$$

This can be proved using Invariant 20.

**Invariants that use Strong Induction in their Proofs:** The proofs of the invariants till now involves using only *weak induction*, *i.e.,* we just assume that the invariants is true only in the current state and prove that any step preserves the in-

variant. We will now state a few invariants that have to all be proved simultaneously using *strong induction*, *i.e.,* we will assume that all the following invariants hold in all states till now and any step preserves the invariants.

**Invariant 29.** If there is a $\langle \mathsf{WaitSt}, l\rangle$-request $\langle c, a, y, x\rangle$, then $dir[l][c] \leq y$.

$$\forall n\ c\ a\ y\ x.\ cRqs.read(n) = \langle\langle \mathsf{WaitSt}, l\rangle, \langle c, a, y, x\rangle\rangle \Rightarrow dir[l][c] \leq y$$

**Invariant 30.** Every slot is "self-consistent", *i.e.,* the directory states of that slot are all compatible and they are all less than or equal to the cache state of that slot.

$$\forall l.\ (\forall i\ j.\ i \neq j \Rightarrow compat(dir[l][i], dir[l][j])) \land \forall i.\ dir[l][i] \leq cs[l]$$

**Invariant 31.** If there is a response from a child to its parent, then the *toState* of that response will be less than the directory state of the parent for that address and child.

$$\forall r \in rssFCTP.\ toState(r) < dir[addr(r)][child(r)]$$

**Invariant 32.** If there is a response from a child to its parent, then the *toState* of that response will be greater than or equal to the cache state of the child for that address.

$$\forall r \in rssFCTP.\ toState(r) \geq (cs[addr(r)] \text{ in cache cache}(r))$$

**Invariant 33.** If there is a response from a parent to its child, then the *toState* of that response will be less than or equal to the directory state of the parent for that address and child.

$$\forall r \in rssFPTC.\ toState(r) \leq dir[addr(r)][child(r)]$$

**Invariant 34.** The directory state of an address for a child in a cache is upgraded iff a response for that address is enqueued into the *toC* buffer and it is downgraded iff a response for that address is popped from the *rsFromC* buffer.

$$\begin{aligned}\forall a\ c.\quad & (dir[a][c] < \mathsf{next}(dir[a][c]) \Leftrightarrow \exists r.\ rsToCSend(r) \wedge \\ & \qquad addr(r) = a \wedge \mathsf{next}(cs[a]) = toState(r) \wedge child(r) = c) \wedge \\ & (dir[a][c] > \mathsf{next}(dir[a][c]) \Leftrightarrow \exists r.\ rsFromCRecv(r) \wedge \\ & \qquad addr(r) = a \wedge \mathsf{next}(cs[a]) = toState(r) \wedge child(r) = c)\end{aligned}$$

**Invariant 35.** The directory state at a parent for a child is greater than or equal to the cache state of that child for the same address.

**Invariant 36.** A request from a child to its parent cannot be extracted by the parent from the *cRqs* buffer before a response enqueued earlier for the same address from that child to the parent has been popped from the *rsFromC* buffer.

**Invariant 37.** If there are multiple requests from a parent to a child for the same address in-flight, then the cache state of the child is $I$.

$$rqsNumFPTC(a) > 1 \Rightarrow cs[a] = I$$

**Invariant 38.** If there is an in-flight response $r$ from a child to its parent and another in-flight request from the parent to the same child for the same address, then $toState(r) = I$.

**Invariant 39.** If a request $r$ from parent to child and a response $r'$ from the same child to the parent for the same address are both in-flight simultaneously, then, in all the states reached after the child sends $r'$, whenever $r$ is in-flight, the cache state at the child for the address is $I$.

**Invariant 40.** There cannot be an in-flight response from a child to its parent and another in-flight response from the parent to the child for the same address.

$$\forall c\ a.\ \mathit{rssNumFCTP}(a) + \mathit{rssNumFPTC}(a) \leq 1$$

We have to prove all these invariants simultaneously by strong induction on a multistep. While some of the invariants does require us to use the strong induction hypotheses, most of the invariants can be proven using the hypotheses for just the current state. The only ones that require the hypotheses for the previous states are Invariants 36 to 40. We state the proofs for those invariants below. Among them, Invariant 36 can be proved easily, and Invariants 37 to 39 can be proven together. Proof of Invariant 40 is complicated and we will give its proof outline next.

This invariant can be violated only if the current step involves a child sending a response to the parent (executed by **Rule**$\langle$*Writeback*$\rangle$ or **Rule**$\langle$*PPropcess*$\rangle$) when there is already a response in-flight from the parent to that child for the same address or involves the parent sending a response to the parent (**Rule**$\langle$*Hit*$\rangle$ or **Rule**$\langle$*Deferred*$\rangle$) when there is already a response in-flight from that child for the same address to the parent.

In **Rule**$\langle$*Writeback*$\rangle$, if a response is sent, then there must be a $\langle\mathsf{WaitV}, l\rangle$-request in the *cRqs* buffer where $l$ is the slot being evicted. But because of Invariant 20, since there is a response for the evicted address from the parent, there must be a $\langle\mathsf{WaitSt}, _\rangle$-request associated for the address being evicted. Because of Invariant 15, it must be a $\langle\mathsf{WaitSt}, l\rangle$-request. Presence of both $\langle\mathsf{WaitV}, l\rangle$- and $\langle\mathsf{WaitSt}, l\rangle$-request in *cRqs* buffer violates Invariant 12.

Let's say the current step is executed by **Rule**$\langle$*PProcess*$\rangle$. Let $r$ be the response from the parent which must be in-flight according to our assumption. To fire **Rule**$\langle$*PProcess*$\rangle$, the child must have received a request, say $r_1$ from the parent

for the same address. Let $r, r_1$ be sent by the parent during steps $t$ and $t_1$. Because of Invariant 26, $t_1$ must occur before $t$ since $r_1$ has been received which $r$ hasn't been received. The directory-wait state at the parent for the child and address in question changes from False to True during step $t_1$ which sends request $r_1$. However, at the beginning of step $t$, which sends response $r$, the directory-wait state must be False, because both the steps that send a response (**Rule**$\langle$*Hit*$\rangle$ and **Rule**$\langle$*Deferred*$\rangle$) require the directory-wait state to be False at the beginning of the step. Thus, a response, say $r_2$, for the same address from the same child must be received at the parent between steps $t_1$ and $t$. By induction hypothesis of Invariant 39, the cache state at the child during the current step must be $I$ because of response $r_2$, in which case **Rule**$\langle$*PProcess*$\rangle$ cannot fire, leading to a contradiction.

Let's say the current step is executed by **Rule**$\langle$*UpgRs*$\rangle$. Let $r$ be the response from the child which must be in-flight according to our assumption for the same address that is being processed by the current step. Let $r$ be sent by the child during step $t$, which can either be **Rule**$\langle$*Hit*$\rangle$ or **Rule**$\langle$*Deferred*$\rangle$. In both the cases, there must be a request $r_1$ sent to the parent from the child. Moreover, since response $r$ has not been received by the parent, step $t$ must occur after the step that sent $r_1$ (let's call this $t_1$), which is received by the parent in the current step. Because of Invariant 19, there must be a ⟨WaitSt, _⟩ request associated with $addr(r_1)$ in the *cRqs* buffer of $child(r_1)$. If step $t_1$ is executed by **Rule**$\langle$*Writeback*$\rangle$, evicting the address in question, there must be a ⟨WaitV, _⟩ request in *cRqs* buffer containing that address. This leads to a contradiction of Invariant 13 since there is a ⟨WaitV, _⟩ request and a ⟨WaitSt, _⟩ request associated with the same address in the *cRqs* buffer. So, step $t_1$ must be **Rule**$\langle$*PProcess*$\rangle$. We can now use an argument similar to the one when the current step is executed by **Rule**$\langle$*PProcess*$\rangle$ to create a contradiction.

**Other Invariants to Prove Invariant 8:** The following invariants can be proved easily using the invariants in the previous section.

**Invariant 41.** Every address is "self-consistent", *i.e.,* the directory states of that address are all compatible and they are all less than or equal to the cache state of that address.

$$\forall a.\ (\forall i\ j.\ i \neq j \Rightarrow compat(dir[a][i], dir[a][j])) \wedge \forall i.\ dir[a][i] \leq cs[a]$$

**Invariant 42.** If $c'$ is an ancestor of $c$, then

$$\forall a.\ cs[a] \text{ in } c\ \leq cs[a] \text{ in } c'$$

**Invariant 43.** If $c$ and $c'$ are such that neither is an ancestor of the other, then

$$\forall a.\ compat(cs[a] \text{ in } c, cs[a] \text{ in } c')$$

**Invariant 44.** When a parent receives a response sent by one of its children, the cache state of the child for the address of the response at the beginning of the step sending the response matches the directory state of the parent for the child cache and address at the beginning of the step receiving the response.

**Invariant 45.** When a child receives a response sent by its parent, the cache state of the child for the address of the response at the beginning of the step receiving the response matches the directory state of the parent for the child cache and address at the beginning of the step sending the response.

**Invariant 46.** No store can take place when any response is in-flight (between any pair of caches), *i.e.,* no step that emits a label $\langle St \rangle$ can fire when any response is in-flight.

**Proving Invariant 8:** This finally brings us to the final invariant that we wanted to prove earlier.

*Proof of Invariant 8.* We will prove this invariant by strong induction on a multistep.

Initially, all and only the memory entries are *clean*, and they contain the latest value for the addresses.

We will assume that in the state reached by a multistep, for some address $a$ in some cache, if $clean(a)$ is true, then $data[a] = latestValue(a)$. We will prove that any new step preserves this invariant.

At the beginning of a transaction, if in cache $c$, $clean(a)$ is true, then it already has the latest value. Moreover, because of Invariant 42, Invariant 43, Invariant 41 and Invariant 35, no other leaf cache has $M$ permission for that address, and thus cache $c$ will continue holding the latest value for address $a$ if $clean(a)$ is true at the end of the multistep.

If $clean(a)$ changes from False to True in cache $c$ during a step, then it must have either downgraded its directory state or upgraded its cache state. It must have either received a response from a child or from its parent, respectively, according to Invariant 28 and Invariant 34.

Consider the case when the parent, for which $clean(a)$ is false, receives a response from its child $c$. By Invariant 44, if the directory state for child $c$ at the parent the changes from $x$ to $y$ when the response is received, then the cache state of the child $c$ must also transform from $x$ to $y$ when the response was sent. Moreover, $x$ must be $M$ since in the parent, $clean(a)$ is false at the beginning of the current step, and if its cache state is $I$ at the beginning of the current step, then the directory-downgrade cannot happen in the current step, because of Invariant 41. Because $clean(a)$ is false at the parent at the beginning of the step, and it receives a downgrade response from the child $c$, it must have downgraded its directory state for child $c$ and address $a$. Thus $cs[a] \neq I$ and $dir[c][a] = x = M$ at the beginning of the step at the

parent. The steps that send a response from child $c$ to parent are **Rule**$\langle$*PProcess*$\rangle$ and **Rule**$\langle$*Writeback*$\rangle$. In both **Rule**$\langle$*PProcess*$\rangle$ and **Rule**$\langle$*Writeback*$\rangle$, when $x$ is $M$, the data is being transferred. Moreover, the child which sends the response satisfies $clean(a)$ state at the beginning of the step (since the directory state of all its children must be less than $M$ to downgrade, and its cache state is $x = M$). So it should contain the latest value by induction hypothesis. Finally, because of Invariant 46, no store can take place during the transfer, thus keeping the latest value.

Consider the case when cache $c$ receives a response from its parent. By Theorem 45, if the cache state of the cache $c$ transforms from $x$ to $y$, then the directory state of the parent for child $c$ must also transform from $x$ to $y$ when it sent the response. Moreover, $x$ must be $I$ since in cache $c$, $clean(a)$ is false at the beginning of the current step and if any of its children is in directory state $M$ for address $a$ at the beginning of the current step, then the cache state cannot upgrade in the current step because of Invariant 41. The steps that send a response to a child are **Rule**$\langle$*Hit*$\rangle$ and **Rule**$\langle$*Deferred*$\rangle$. In both **Rule**$\langle$*Hit*$\rangle$ and **Rule**$\langle$*Deferred*$\rangle$, when $x$ is $I$, the data is being transferred. Moreover, the parent which sends the response satisfies $clean(a)$ at the beginning of the step (since the directory state of all its children must be less than $M$ as they are all compatible with the requested upgraded by cache $c$, and its cache state is greater than $I$ since it can send an upgrade response to the child $c$). So it should contain the latest value by induction hypothesis. Finally, because of Invariant 46, no store can take place during the transfer, thus keeping the latest value, hence proving the theorem. □

## 8.2 Deadlock-freedom in a Hierarchy of Coherent Caches

In Section 8.1, we focused on the proof of soundness of the cache coherence protocol. It is essential to prove other properties of this protocol, like deadlock-freedom and livelock-freedom. Deadlock-freedom is especially important because even a system which does not have any rules, and hence does not undergo any steps, is sound with respect to any specification. We want to disallow such spurious systems. We first prove that the cache-coherence protocol is such that in any state reached from the initial state by the multistep, at least one Bluespec rule can fire.

**Theorem 15.** There is always some Bluespec rule that can fire in $M'_{\text{cache}}$ as long as there are requests from the processors in *cRqs* buffers of the L1 caches, and the *toC* buffers of L1 caches that have non-empty *cRqs* buffers are not full.

*Proof.* Consider the case when any response from a child to its parent is present in the *rsFromC* buffer of cache $c$. Then, **Rule**$\langle$*DwnRs*$\rangle$ for cache $c$ can fire.

Consider the case when no response from any child to its parent is present in any *rsFromC* buffer of any cache. If there is a response from child to parent in the network or in *rsToP* buffers, then the step transferring this message into the network or the destination *rsFromC* buffer can fire.

Now consider the case when there are no responses in-flight from any child to its parent.

Consider a cache $c$ such that in all the descendants of this cache (including itself), there are no messages in-flight from the descendants to any of its children, but cache $c$ itself has a request from its parent in-flight. If the request in the *toC* buffer of its parent, and the network is not full, then the rule that transfers the message from the *toC* buffer to the network will fire. Instead, if the request is in the network and the *fromP* buffer of $c$ is not full, then the rule that transfers the message from the

network to the *fromP* buffer will fire. And instead, if the request is in the *fromP* buffer but the *pRqs* buffer is not full, **Rule**⟨*PTransfer*⟩ will fire. Finally, consider the case when there are messages in the *pRqs* buffer.

Consider the sub-case when there is a ⟨WaitSt, _⟩ request $r$ in the *cRqs* buffer of cache $c$ such that the function *completed* returns true because of this request, *i.e.,* $cs[addr(r)] \geq toState(r)$. If $c$ is an L1 cache, then $toC$ is not full, allowing **Rule**⟨*LdDeferred*⟩ or **Rule**⟨*StDeferred*⟩ to fire. If $c$ is not an L1 cache, then the $toC$ buffer is empty as there are no messages from cache $c$ to its children. Either **Rule**⟨*Deferred*⟩ can fire (if the directory states of the rest of the children are compatible) or **Rule**⟨*DwnRq*⟩ will fire, if some other child's directory state is not compatible with $toState(r)$.

Consider the sub-case when there is no ⟨WaitSt, _⟩ request $r$ in the *cRqs* buffer of cache $c$ such that $cs[addr(r)] \geq toState(r)$, *i.e.,* the function *completed* returns false. In this sub-case, since there is a request in the *pRqs* buffer, one of **Rule**⟨*Drop*⟩, **Rule**⟨*PProcess*⟩ or **Rule**⟨*DwnRqP*⟩ will fire. If the cache state at the cache is lower than the *toState* of the request in the *pRqs* buffer, then **Rule**⟨*Drop*⟩ will fire. If the cache state is higher than the *toState* of the request, but the directory states of all its children are lower, then **Rule**⟨*PProcess*⟩ can fire. Firing **Rule**⟨*PProcess*⟩ requires that $rsToP$ is not full, which is guaranteed because $rsToP$ is empty. It also requires that any ⟨WaitSt, $l$⟩ request $r$ in the *cRqs* buffer of cache $c$ is such that $cs[l] < toState(r)$, which is guaranteed in this sub-case. If the cache state is higher than *toState* of the request and the directory state of some child is higher, then **Rule**⟨*DwnRqP*⟩ will fire for that child. Firing **Rule**⟨*DwnRqP*⟩ requires that $toC$ is not full, which is guaranteed since there are no messages in-flight from $c$ to its children. It also requires the same condition about ⟨WaitSt, $l$⟩ requests as for firing **Rule**⟨*PProcess*⟩, which is guaranteed in this sub-case.

Now consider the case when there are no messages from any parent to any child

anywhere in the system. Since there are no requests from parent to children and no responses from children to parent, the directory-wait states for all caches and all addresses are unset because of Invariant 47 which we state below.

**Invariant 47.** If the directory-wait state is set for an address and child, then there must either be a request from that parent for the same address or a response from that child for the same address.

$$\forall a\ c.\ \mathit{dirw}[a][c] \Rightarrow \mathit{rqsFPTC}(a,c) + \mathit{rssFCTP}(a,c) > 0$$

Proving this invariant is easy and we omit its proof.

Consider a cache $c$ such there there are no in-flight requests from any of its ancestors (including itself) to the ancestor's parent and there is an in-flight request from a child of $c$ to $c$. If there is a request in the *rqToP* buffer of the child of $c$ but the network is not full, then the rule that transfers the message from the *rqToP* buffer to the network will fire. Instead, if a request is in the network and the *rqFromC* buffer at $c$ is not full, then the rule that transfers the message from the network to the *rqFromC* buffer will fire. And if a request is in the*rqFromC* buffer and the *cRqs* buffer is not full, then **Rule**⟨*CTransfer*⟩ will fire. Finally, consider the case when there are messages in the *cRqs* buffer.

Consider the case when $c$ is an L1 cache. Consider the sub-case when there are ⟨WaitSt, _⟩-requests in the *cRqs* buffer. If the request from the processor requires a cache state upgrade, then **Rule**⟨*UpgRq*⟩ can fire as there are no requests in-flight from $c$ to its parent and hence the *rqToP* buffer is empty. Otherwise either **Rule**⟨*LdDeferred*⟩ or **Rule**⟨*StDeferred*⟩ can fire depending on the request (since the corresponding *toC* buffer is not full). Now consider the sub-case when there are only ⟨WaitV, _⟩-requests in the *cRqs* buffer. Then **Rule**⟨*Writeback*⟩ can fire as the *rsToP* buffer is empty. Finally, consider the sub-case when there are only Init-requests in the

*cRqs* buffer. There are no requests in the *pRqs* buffer, and no waiting requests in *cRqs* buffer. So *searchAddr* returns $\epsilon$ for both *cRqs*- and *pRqs*-search. Function *searchSlot* also returns $\epsilon$. Moreover, the response buffer *toC* is empty. Thus, one of **Rule**⟨*LdHit*⟩, **Rule**⟨*StHit*⟩, **Rule**⟨*MissByState*⟩ or **Rule**⟨*MissByLine*⟩ can fire, depending upon the request.

Consider the case when $c$ is not an L1 cache. Consider the sub-case where there are ⟨WaitSt, _⟩-requests in the *cRqs* buffer. If the request requires a cache state upgrade, then **Rule**⟨*UpgRq*⟩ can fire as there are no requests in-flight from $c$ to its parent and hence the *rqToP* buffer is empty. If the directory states of the other children are not compatible, then **Rule**⟨*DwnRq*⟩ can fire since the *toC* buffers are all empty. Finally, if the cache state is high enough, and the other directory states are compatible, then **Rule**⟨*Deferred*⟩ can fire since directory-wait state is not set in this case.

Now consider the case when there are only ⟨WaitV, _⟩-requests in *cRqs* buffer. If the directory state of any child is not $I$, then **Rule**⟨*DwnRqEvict*⟩ can fire since *toC* buffer is empty and the directory-wait state is not set. If the directory state of all the children is $I$, then **Rule**⟨*Writeback*⟩ can fire since *rsToP* buffer is empty.

Finally, consider the case when there are only Init-requests in the *cRqs* buffer. The directory-wait state is not set, there are no requests in the *pRqs* buffer, and there are no waiting requests in the *cRqs* buffer. So *searchAddr* returns $\epsilon$ for both *cRqs*- and *pRqs*-search. Function *searchSlot* also returns $\epsilon$. Moreover, the buffer *toC* is empty. If the address corresponding to an Init-request is $a$ and if the directory state of $a$ is greater than the *fromState* of the Init-request, there there must be a response from the same child as the Init-request because of the following invariant which we prove below.

**Invariant 48.** If a request from a child to its parent is in-flight where the *fromState* of the request is less than the current directory state of the parent for that address and child, then there must be an in-flight response for that address from the child

with *toState* equal to the *fromState* of the request.

$$\forall r \in rqsFCTP.\ dir[child(r)][addr(r)] > fromState(r) \Rightarrow$$

$$\exists r' \in rssFCTP.\ child(r) = child(r') \wedge addr(r') = addr(r) \wedge toState(r') = fromState(r)$$

But no responses from children are in-flight in this case. This ensures that the directory state is less than the *fromState* of any Init-request for the same address. Thus, one of **Rule**⟨*Hit*⟩, **Rule**⟨*MissByState*⟩ or **Rule**⟨*MissByLine*⟩ can fire, depending upon the request. □

We now prove Invariant 48.

*Proof of Invariant 48.* The directory state cannot be $I$ since it is greater than the *fromState* of the request $r$ which is in-flight from the child. So, the directory state must have undergone some state transition (according to Invariant 28).

Consider the state when the last directory state change for $addr(r)$ and cache($r$) is a downgrade. By Invariant 34, there must a response, say $r_1$, received by the parent during the downgrade. This has to be sent by the child before sending request $r$ so as to not violate Invariant 36. If the child has undergone any cache state change after sending $r_1$, then it must have sent or received a response, according to Invariant 28. Let it have received a response. This response cannot be sent by the parent before receiving $r_1$ so as to not violate Invariant 40. It cannot be sent after receiving $r_1$ since no directory state transition happened after sending $r_1$. So the child must instead have sent a response. Consider the last response, say $r_2$, sent by the child. It must be the case that $toState(r_2) = fromState(r)$ since $r_2$ is the last response sent by the child before sending $r$. Response $r_2$ cannot have been received by the parent because of Invariant 34 since the directory state has not changed since sending $r_1$. This satisfies the invariant.

A similar analysis can be done for the case when the last directory state change for $addr(r)$ and cache$(r)$ is an upgrade. By Invariant 34, there must a response, say $r_1$, sent by the parent to the child during the upgrade. This has to be received by the child before sending request $r$ so as to not violate Invariant 18. If the child has undergone any cache state change after receiving $r_1$, then it must have sent or received a response, according to Invariant 28. Let it have received a response. This response cannot be sent by the parent before sending $r_1$ so as to not violate Invariant 22. It cannot be sent after sending $r_1$ since no directory state transition happened after sending $r_1$. So the child must instead have sent a response. Consider the last response, say $r_2$, sent by the child. It must be the case that $toState(r_2) = fromState(r)$ since $r_2$ is the last response sent by the child before sending $r$. Response $r_2$ cannot have been received by the parent since it cannot be received before receiving $r_1$ so as to not violate Invariant 21 and because of Invariant 34 since the directory state has not changed since receiving $r_1$. This satisfies the invariant. □

We can now state and prove the deadlock-freedom property of the cache hierarchy.

## 8.3 Livelock-freedom in a Hierarchy of Coherent Caches

In Section 8.2, we proved that the cache-coherence protocol is deadlock-free, *i.e.,* if there is a request and space to send the response, then some rule in the cache hierarchy will fire. While this is an important property, this alone is not enough to claim that a cache-coherence protocol is correct. For example, consider a protocol with just one unguarded rule which just increments a counter. It is easy to see that this protocol is sound, since it does not call any method, or have any methods defined in it (and hence does not create any labels). It is also easy to see that this rule can always fire, thus satisfying our definition of deadlock-freedom.

In order to really prove that a protocol is correct, one has to show that some form

of "forward progress" is made. That is, when requests from processors are present, and there is space to send the corresponding responses, then eventually some response will be sent. This is exactly what we are going to prove in this section.

While the proofs of soundness and deadlock-freedom were mechanically verified in the Coq proof assistant, the final proof for livelock-freedom has not been mechanically verified. There is no inherent difficulty in implementing this proof in Coq; one has to describe a total order for the states reached by the system and prove that any step increases the state with respect to this order.

We will now state and prove the livelock-freedom property.

**Theorem 16.** As long as there are requests from the processors in *cRqs* buffers of the L1 caches, and the *toC* buffers of L1 caches that have non-empty *cRqs* buffers are not full, a response for a request will eventually be enqueued into the corresponding *toC* buffer.

*Proof.* As discussed briefly earlier, the basic idea for the livelock-freedom proof is to define an ordering between states and prove that whenever a step takes place, it moves the state up the order. Each request from processors goes through several *stages* before it is popped and a corresponding response is enqueued. The order for states is defined in terms of the sequence of stages that each request goes through. The initial stage of a request is when it is enqueued into a *rqFromC* buffer of an L1 cache, and the final stage is when it is dequeued from the *rqFromC* buffer of the L1 cache and enqueued into the corresponding *toC* buffer. The number of stages that any request goes through is finite. If we show that each step takes at least one request forward to its next stage, while keeping the other requests in the same stage, then since the number of stages that any request goes though is finite, eventually a response will be enqueued into the corresponding *toC* buffer since the last stage corresponds to a response in the *toC* buffer. We will now describe the stages that any request goes through.

We will first define the *local stages* that any request goes through. These stages are local because they are based on the local state in the cache where the request is present.

We will first show the ordering of the various stages that a request $r$ from a processor to an L1 cache $c$ goes through in Figure 8-1, where the topmost is the first stage and the bottommost is the last stage.

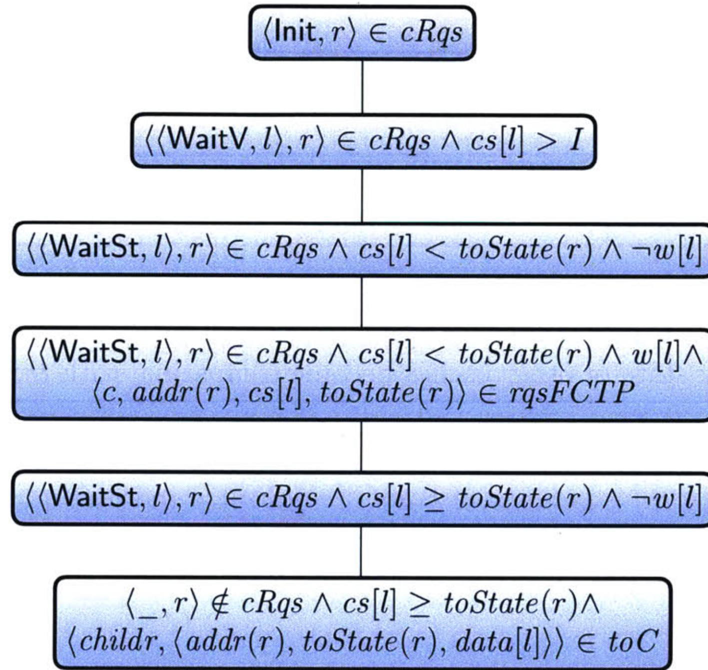



Figure 8-1: Stages for a request $r$ from a processor to an L1 cache $c$

Next we will show the ordering of the various stages that a request $r$ from a cache to its parent $c$ goes through in Figure 8-2, starting with a request in the $rqToP$ buffer of the child and ending with a response in the $toC$ buffer of the parent $c$. Again, the topmost is the first stage and the bottommost is the last stage.

The two yellow boxes represent several stages that the request $r$ goes through. The cache state of slot $l$ and the augmented state for the request $r$ remain the same in all stages in both the boxes while the cross-product of the directory state and directory-wait state pairs for each cache in slot $l$ changes.

The first yellow box represents the stages that a request goes through for evicting

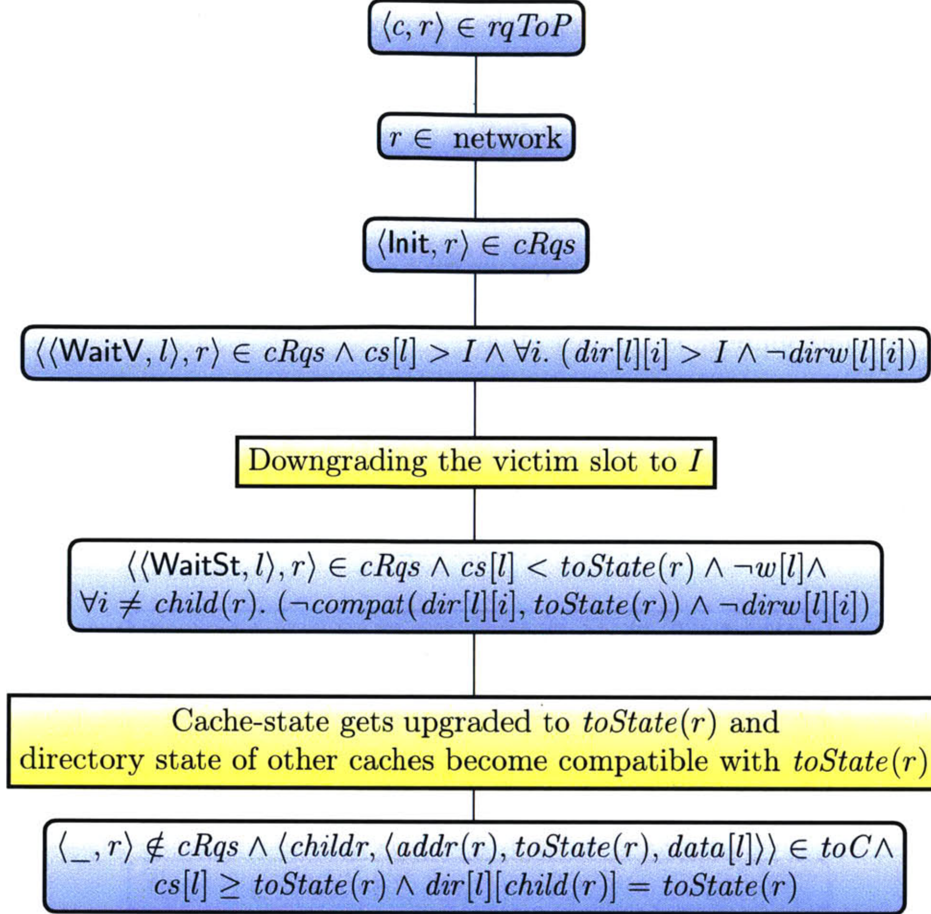



Figure 8-2: Stages for a request $r$ from a cache to its parent $c$ (which is not memory)

a cache line when directory states become $I$. The partial order between stages within this box is given in Figure 8-3, where $c_1 \ldots c_n$ are the children of $c$.

The second yellow box represents upgrading the cache state of $c$ and downgrading the directory states of all children other than the one which sent request $r$ (so that their directory states are compatible with the requested upgrade). Its partial order is given in Figure 8-4, where $c_1 \ldots c_n$ are the children of $c$ which are not $child(r)$. We give the partial order for the case when the requested upgrade is $M$. If the requested upgrade is $S$, then the stages where $dir[l][c_i] = I$ are not present in the partial order, and $cs[l]$ gets upgraded to $S$ instead of $M$.

Next we will show the ordering of the various stages that a request $r$ from a child of memory goes through in Figure 8-5, starting with a request in the $rqToP$ buffer of

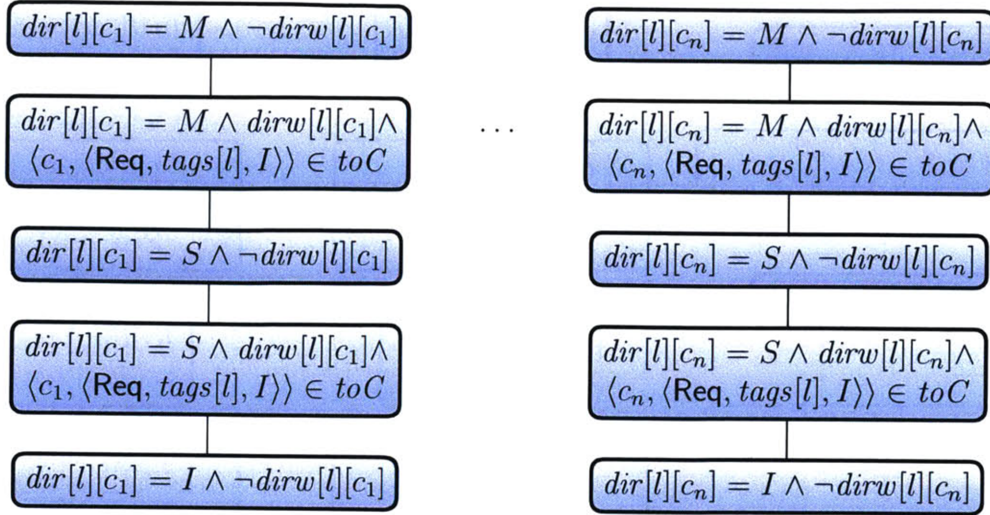



Figure 8-3: Partial order for stages representing evicting a cache line

the child and ending with a response in the $toC$ buffer of the memory. This is very similar to the ordering for Figure 8-2 except for the absence the parent.

The yellow box represents downgrading the directory states of all children other than the one which sent request $r$ (so that their directory states are compatible with the requested upgrade). Its partial order is given in Figure 8-6, where $c_1 \ldots c_n$ are the children of $c$ which are not $child(r)$. As before, we give the partial order for the case when the requested upgrade is $M$. The figure has to be changed appropriately if the requested upgrade is $S$.

We will now give the order of stages that a request $r$ from parent to a child goes through in Figure 8-7.

As earlier, the yellow box represents several stages that the request $r$ goes through. The partial order between stages within the second yellow box is given in Figure 8-8. Caches $c_1 \ldots c_n$ are the children of the cache that receives request $r$. The figure uses a new predicate $waited(a, c_i)$, which denotes if $dirw[a][c_i]$ has ever been set after request $r$ has been received. As long as $dirw[a][c_i]$ has not been set earlier, changes in directory state do not affect the partial order. Similarly to our treatment of directory state changes for processing a request in the $cRqs$ buffer, we show only the case when

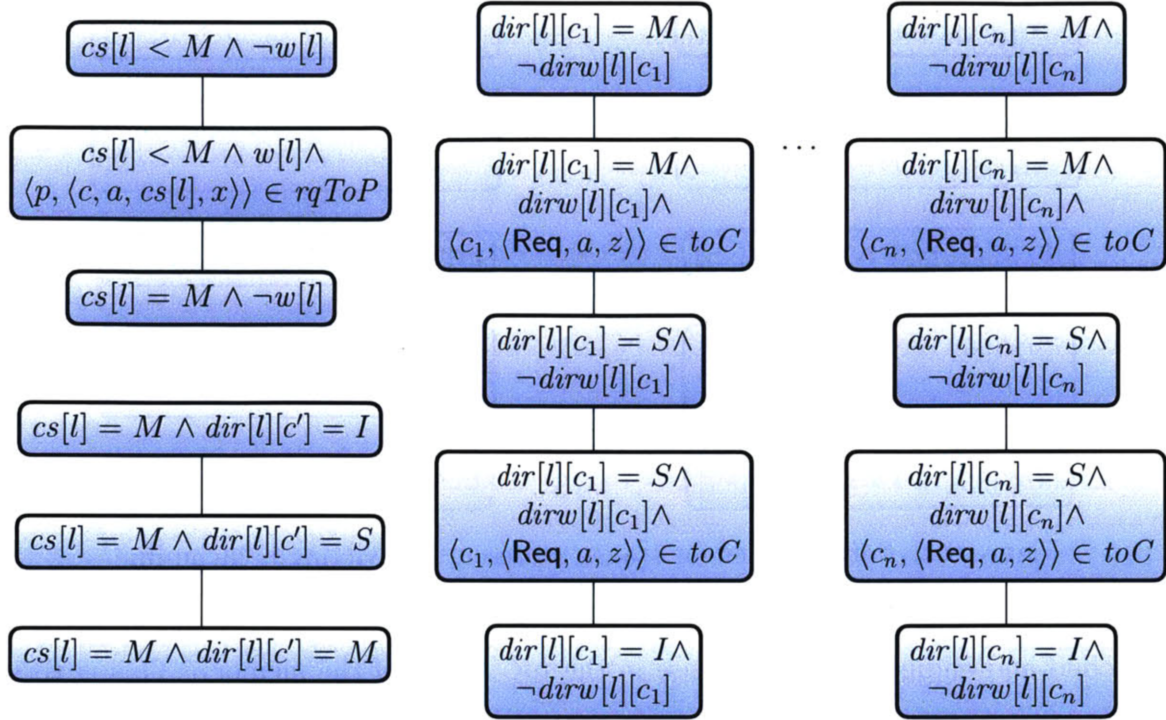



Figure 8-4: Partial order for stages representing upgrading the cache state and downgrading the other children, for a $\langle \mathsf{WaitSt}, l \rangle$-request $r = \langle c', a, y, M \rangle$ in the *cRqs* buffer of an internal cache

the downgrade being requested is to $I$. The stages containing $dir[l][c_i] = I$ will be absent in case the downgrade being requested is $S$.

As mentioned before, the partial orders that we have defined in Figures 8-1 to 8-8 are local to the cache where the request is present. A request in the *cRqs* buffer is created due to some request in the *cRqs* buffer of the child sending the request, which recursively leads to a processor request in the L1 cache. A request in the *pRqs* buffer is created due to some request either in the *cRqs* buffer of the parent, or in the *pRqs* buffer of the parent. At the memory, every request is created due to some request in the *cRqs* buffer. So, even requests in the *pRqs* buffer are eventually created because of a processor request in the L1 cache. We use this property to associate every stage defined in Figures 8-1 to 8-8 to one or more requests from the processor in the L1 cache, which we call the *origin requests*. We will prove that any step moves up the stage of some origin request and does not move down the stage of any other origin

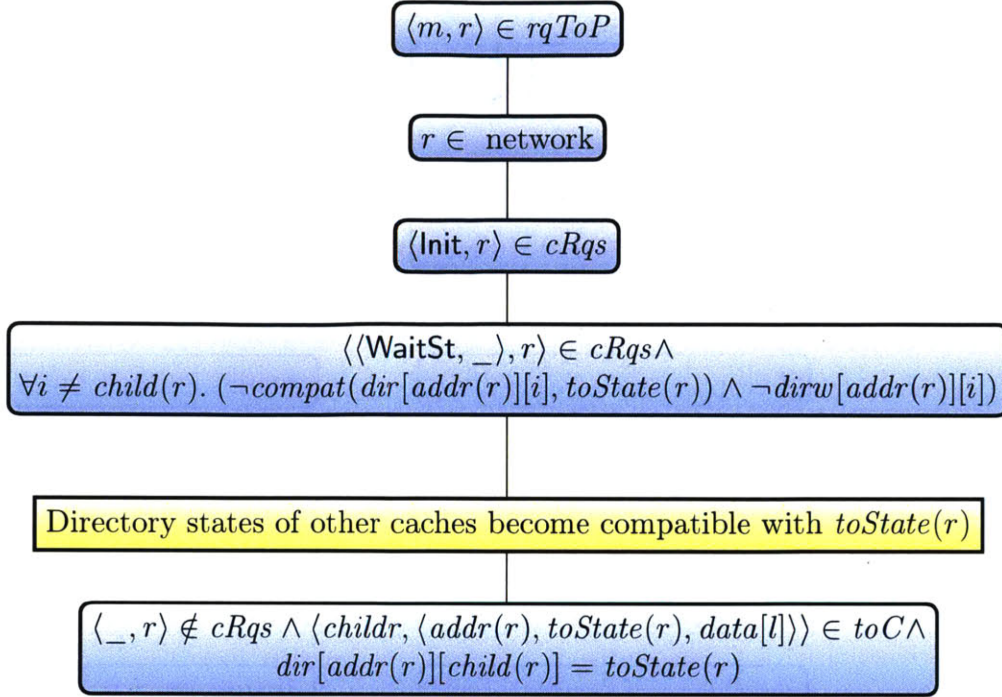



Figure 8-5: Stages for a request $r$ from a cache to memory $m$

request. If this is true, then some origin request makes forward progress in every step (since the last stage of an origin request sends an appropriate response back to the appropriate processor). And since there are only a finite number of stages, we will be able to prove that a response will eventually be sent back to the processor.

We are left with proving that any step moves up the stage of some origin request and does not move down the stage of any other origin request. It is easy to see that every step moves up the stage of some origin request. But, because of the conflict between a request in a *cRqs* buffer and a request in a *pRqs* buffer, it is not obvious if no other origin request moves down the stage because of a step. There are two such conflicts:

1. The cache state downgrades because of a request in the *pRqs* buffer, but another request in the *cRqs* buffer requires the cache state to be upgraded. So, the origin request for the latter request can potentially move down the stage. Similarly, the request in the *pRqs* buffer can cause a directory state to downgrade when

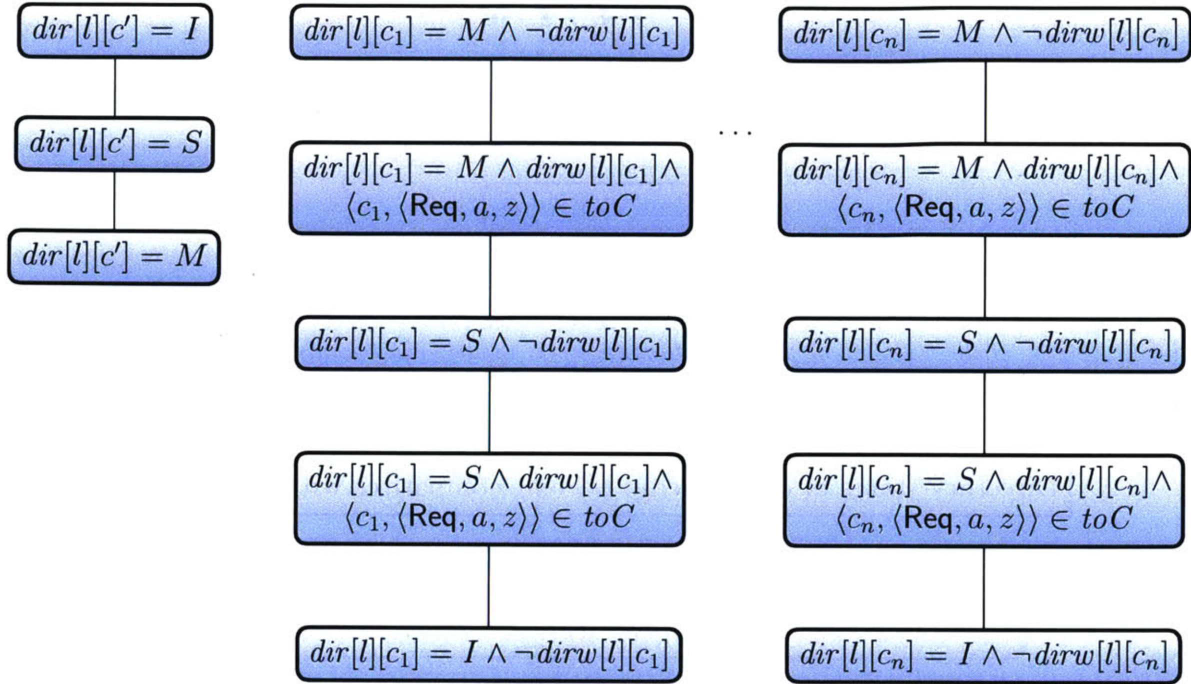



Figure 8-6: Partial order for stages representing upgrading the cache state and downgrading the other children, for a WaitSt-request $r = \langle c', a, y, M \rangle$ in the $cRqs$ buffer of memory

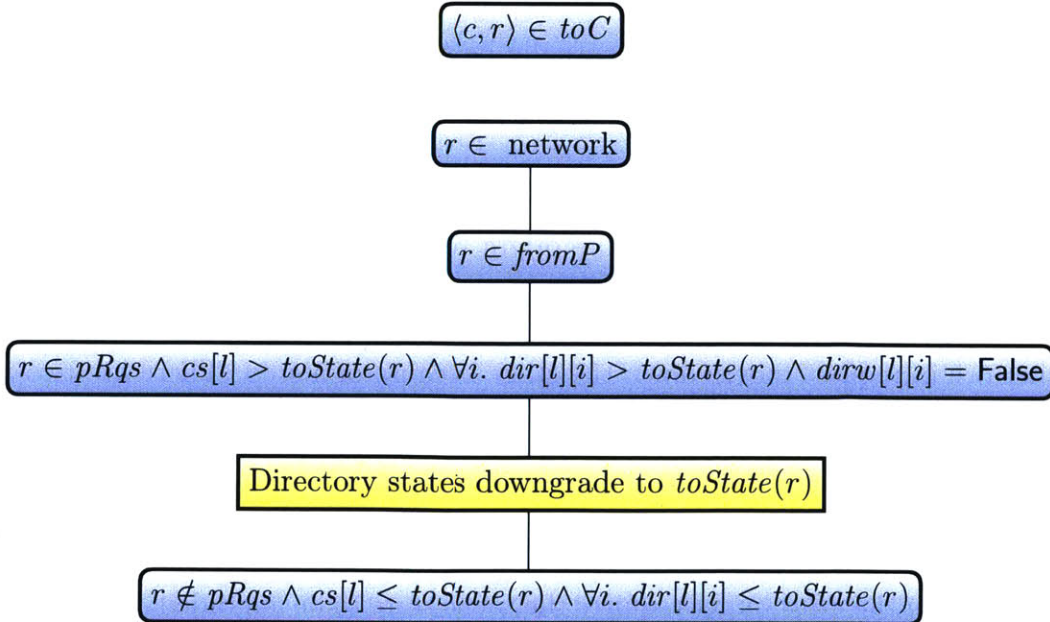



Figure 8-7: Stages for a request from a parent to its child $c$

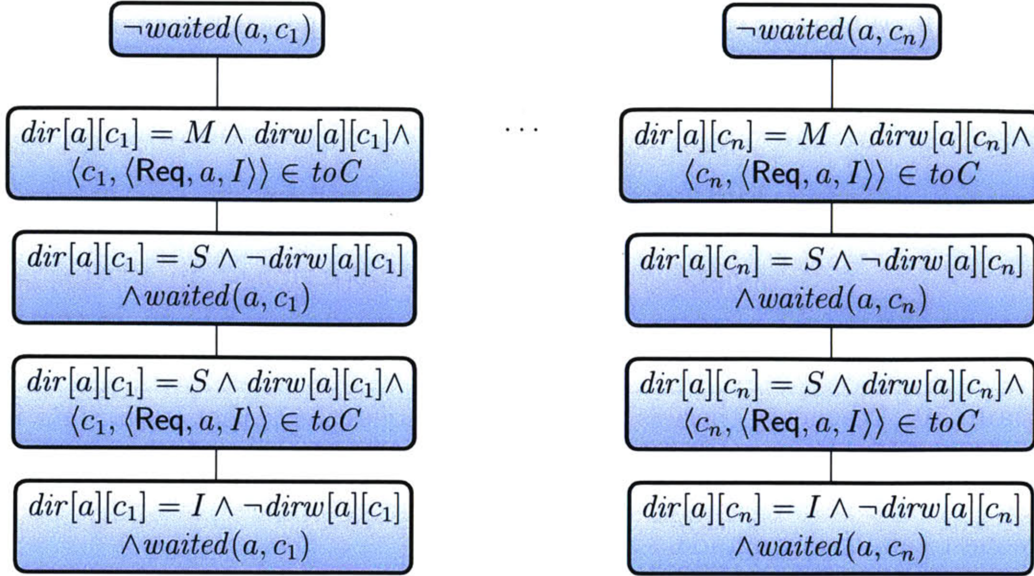



Figure 8-8: Partial order for stages representing downgrading a cache line for processing request $r = \langle \mathsf{Req}, a, I\rangle$ from the parent.

there is a request in the *cRqs* buffer requiring the same directory state to be upgraded.

2. The cache state upgrades because of a request in the *cRqs* buffer, but another request in the *pRqs* buffer requires the cache state to be downgraded. Again, the origin request for the latter request can move down the stage. Similarly, the request in the *cRqs* buffer can cause a directory state to upgrade when there is a request in the *pRqs* buffer requiring the same directory state to be downgraded.

The way the stages are set up is to avoid these conflicts. With respect to the first conflict, it is acceptable for the cache state and directory state to downgrade even when a request $r$ is present in the *cRqs* buffer for the same address, as long as the cache state for that address has not yet been upgraded to $toState(r)$ (or higher). This can be seen, for example, in Figure 8-4; moving up the stage requires that once the cache state has been upgraded to $toState(r)$, neither the cache state nor the directory state for the child making the request cannot downgrade. We will show that this is indeed the case using Invariants 49 and 50, which we state below and prove later.

**Invariant 49.** Whenever the *toState* of a $\langle \mathsf{WaitSt}, _ \rangle$-request is less than or equal to the corresponding cache state, the cache state cannot downgrade.

$$\forall r\ n.\ \langle\langle \mathsf{WaitSt}, _ \rangle, r\rangle = cRqs.read(n) \Rightarrow$$
$$toState(r) \leq cs[addr(r)] \Rightarrow \mathsf{next}(cs[addr(r)]) \geq cs[addr(r)]$$

**Invariant 50.** Whenever there is a $\langle \mathsf{WaitSt}, _ \rangle$-request such that its *toState* is less than or equal to the corresponding cache state, the directory state for the child sending that request cannot downgrade.

$$\forall r\ n.\ \langle\langle \mathsf{WaitSt}, _ \rangle, r\rangle = cRqs.read(n) \wedge toState(r) \leq cs[addr(r)] \Rightarrow$$
$$\mathsf{next}(dir[addr(r)][child(r)]) \geq dir[addr(r)][child(r)]$$

With respect to the second conflict, it is easy to show that that once a request is present in the *pRqs* buffer, the corresponding cache state cannot be upgraded. The only way to upgrade a cache state is by receiving a response from the parent by Invariant 28. But no response can be received if there is a request in the *pRqs* buffer for the same address.

However, the directory state can actually be upgraded even when there is a request in the *pRqs* buffer. This is because there can be a $\langle \mathsf{WaitSt}, _ \rangle$-request $r$ for the same address such that $cs[addr(r)] \geq toState(r)$, in which case **Rule**⟨*(Ld/St)Deferred*⟩ can fire, upgrading the directory state. But this is acceptable because of the way the stages are set up for a request in the *pRqs* buffer. As can be seen in Figure 8-8, a directory state can upgrade as long as the directory-wait state has never been set ($\neg waited$) since a request for the same address was enqueued in the *pRqs* buffer. But once the directory-wait state has been set (*i.e.*, *waited* is true), the directory-wait state should not upgrade anymore to preserve the forward movement across stages.

We prove that this is indeed the case in Invariant 51 stated below.

**Invariant 51.** If a request $r$ is present in the *pRqs* buffer, and if the directory-wait state for that address for some child is set, then the directory state for the same address and child can no longer be upgraded as long as $r$ is present in the *pRqs* buffer.

$$\forall c.\ r \in \mathit{pRqs.all}.\ \mathit{waited}(\mathit{addr}(r), c) \Rightarrow \mathsf{next}(\mathit{dir}[\mathit{addr}(r)][c]) \leq \mathit{dir}[\mathit{addr}(r)][c]$$

□

We are thus left with stating and proving Invariants 49 to 51. Invariant 49 is straightforward to prove.

To prove the above Invariants 50 and 51, we need a few more invariants which are stated below. Of these, the proof of Invariant 53 is more complicated and we give its full proof.

**Invariant 52.** If there is a $\langle \mathsf{WaitV}, _ \rangle$-request for an address $a$, then no slot contains address $a$.

$$\forall r\ n.\ \mathit{cRqs.read}(n) = \langle\langle \mathsf{WaitV}, _ \rangle, r\rangle \Rightarrow \mathit{searchTags}(\mathit{cs}, \mathit{tags}, \mathit{addr}(r)) = \epsilon$$

**Invariant 53.** Whenever an address is present in the *pRqs* buffer, then no request $r$ in the *cRqs* buffer can become a $\langle \mathsf{WaitSt}, _ \rangle$-request with $\mathit{cs}[\mathit{addr}(r)] \geq \mathit{toState}(r)$.

$$\begin{aligned}\forall s\ r\ n.\ \mathit{cRqs.read}(n) = \langle s, r\rangle \wedge s \neq \langle \mathsf{WaitSt}, _ \rangle \wedge \\ \mathit{searchAddr}(\mathit{pRqs.all}, \mathit{addr}(r)) \neq \epsilon \Rightarrow \\ \neg(\mathsf{next}(\mathit{cRqs.read}(n)) = \langle\langle \mathsf{WaitSt}, _ \rangle, r\rangle \wedge \mathit{cs}[\mathit{addr}(r)] \geq \mathit{toState}(r))\end{aligned}$$

*Proof.* The only steps that convert an $\mathsf{Init}$-request $r$ into a waiting request (either

a $\langle \mathsf{WaitV}, _ \rangle$- or a $\langle \mathsf{WaitSt}, _ \rangle$-request) are executed by **Rule**$\langle$*MissByState*$\rangle$ and **Rule**$\langle$*MissByLine*$\rangle$. Neither can occur when $addr(r)$ is present in the *pRqs* buffer. Thus, the only way this theorem can be violated is if

1. $r$ was a $\langle \mathsf{WaitV}, _ \rangle$-request before $addr(r)$ was present in the *pRqs* buffer and after a request containing $addr(r)$ was enqueued into the *pRqs* buffer, it eventually changed to a $\langle \mathsf{WaitSt}, _ \rangle$-request with $cs[addr(r)] \geq toState(r)$ or

2. $r$ was a $\langle \mathsf{WaitSt}, _ \rangle$-request before $addr(r)$ was present in the *pRqs* buffer and the cache state got upgraded to a value greater than or equal to $toState(r)$ after a request containing $addr(r)$ was enqueued into the *pRqs* buffer.

In the first case, as long as the request $r$ is in $\langle \mathsf{WaitV}, _ \rangle$ augmented state, by Invariant 52, $cs[addr(r)] = I$ since no slot contains $addr(r)$. Thus, the cache state must have upgraded after $r$ becomes a $\langle \mathsf{WaitSt}, _ \rangle$-request, which must happen only after $addr(r)$ is present in the *pRqs* buffer. Therefore, in both the cases, the cache state must have upgraded after a request, say $r'$ for $addr(r)$ is enqueued into the *pRqs* buffer. The upgrade is accompanied by receiving a response, say $r''$ from its parent, as per Invariant 28.

Since request $r'$ is present in the *pRqs* buffer before receiving response $r''$, it must have been sent by the cache's parent before sending $r''$ because of Invariant 25. The directory-wait state at the parent is set right after sending $r'$ and is false right before sending $r''$ (because both **Rule**$\langle$*Hit*$\rangle$ and **Rule**$\langle$*Deferred*$\rangle$ that send a response from the parent to the cache require that the directory-wait state is false at the beginning of the respective Bluespec rules). The parent must have received a response from the cache between sending $r'$ and sending $r''$. Because of Invariant 39, the cache state of the cache should be $I$ as long as $r'$ is present in the *pRqs* buffer, leading to a contradiction. □

We will now prove Invariants 50 and 51.

*Proof of Invariant 50.* We will assume that there is a $\langle$WaitSt, _$\rangle$-request obeying the requirement but that directory state downgrades during the current step and prove this invariant by contradiction. The directory state can be downgraded only on receiving a response, say $r_1$, from the corresponding child according to Invariant 34. This response must be sent from the child during the steps executed by **Rule**$\langle$*Writeback*$\rangle$ or **Rule**$\langle$*PProcess*$\rangle$. Since there is a $\langle$WaitSt, _$\rangle$-request, $r$, from the child during the current step when $r_1$ is received at the child, $r$ must have been sent before $r_1$ was sent because of Invariant 36. Moreover, the response corresponding to request $r$ cannot be received between sending $r$ and $r_1$ so as to not violate Invariant 18. Thus, there is an in-flight request from the child to the parent during the step that sends response $r_1$, which means, by Invariant 19, there is a $\langle$WaitSt, $l\rangle$-request for the $addr(r_1)$ when $r_1$ is sent. If the step sending $r_1$ is **Rule**$\langle$*Writeback*$\rangle$, then there will be a $\langle$WaitV, $l\rangle$-request containing $addr(r_1)$, which contradicts Invariant 12. Thus, the step sending $r_1$ must be **Rule**$\langle$*PProcess*$\rangle$. So, the child must have received a downgrade request, say $r_2$ from the parent during **Rule**$\langle$*PProcess*$\rangle$.

If the parent received a response from the child between sending request $r_2$ and receiving response $r_1$, then by Invariant 39, the cache state of the child during step executed by **Rule**$\langle$*PProcess*$\rangle$, when response $r_1$ is sent, must be $I$, leading to a contradiction. Thus, the directory states remain the same between the beginning of the step sending request $r_2$ and the beginning of the step receiving response $r_1$ because of Invariant 34.

The only steps that send request $r_2$ to the child are executed by **Rule**$\langle$*DwnRqEvict*$\rangle$, **Rule**$\langle$*DwnRq*$\rangle$ and **Rule**$\langle$*DwnRqP*$\rangle$.

If **Rule**$\langle$*DwnRqEvict*$\rangle$ sent request $r_2$, there is some $\langle$WaitV, $l\rangle$-request $r'$ where slot $l$ contains $addr(r_2)$. Request $r'$ remains a $\langle$WaitV, $l\rangle$-request at the beginning of the step receiving response $r_1$ since changing the augmented state requires firing of **Rule**$\langle$*Writeback*$\rangle$ which in turn requires the directory states to have changed to $I$

between sending $r_2$ and receiving $r_1$. If there is another ⟨WaitSt, _⟩-request for the same address, then by Invariant 15, it must point to the same slot, thus violating Invariant 12. This leads to a contradiction.

If **Rule**⟨*DwnRq*⟩ sent request $r_2$, there is some ⟨WaitSt, _⟩-request $r'$ for the same slot containing $addr(r_2)$. Request $r'$ should be from a different child than $child(r)$ since a downgrade request is not sent to the same child requesting an upgrade in this rule. Request $r'$ remains a ⟨WaitSt, _⟩-request at the beginning of the step receiving response $r_1$ since changing the augmented state requires firing of **Rule**⟨*Deferred*⟩ which in turn requires the directory state to have downgraded for $child(r)$ between sending $r_2$ and receiving $r_1$. By Invariant 13, there cannot be another ⟨WaitSt, _⟩-request $r$ for the same address, leading to a contradiction.

If **Rule**⟨*DwnRqP*⟩ sent request $r_2$, then $addr(r)$ must be present in the *pRqs* buffer. By Invariant 53, request $r$ must have become a ⟨WaitSt, _⟩-request with $cs[addr(r)] \geq toState(r)$ before the request containing $addr(r)$ was enqueued into the *pRqs* buffer. This prevents firing of **Rule**⟨*DwnRqP*⟩, since the rule explicitly checks that there are no ⟨WaitSt, _⟩-requests whose cache states are greater than or equal to the requests' *toState* using the *completed* function. This again leads to a contradiction, hence proving the invariant. □

*Proof of Invariant 51.* The only steps that upgrade the directory state are executed by **Rule**⟨*Hit*⟩ and **Rule**⟨*Deferred*⟩ because of Invariant 34. Out of these, if $r$ is present in the *pRqs* buffer, then **Rule**⟨*Hit*⟩ cannot fire for a request in *cRqs* corresponding to the same address because *searchAddr* applied on *pRqs* will return non-$\epsilon$. If we prove that **Rule**⟨*Deferred*⟩ cannot fire under the appropriate assumptions, then the invariant holds.

For **Rule**⟨*Deferred*⟩ to fire, there has to be a ⟨WaitSt, _⟩-request, say $r'$, in the *cRqs* buffer for the same address as $r$, where $cs[addr(r')] \geq toState(r')$. By Invariant 53, request $r'$ must have become a ⟨WaitSt, _⟩-request with $cs[addr(r')] \geq$

$toState(r')$ before $r$ is enqueued into the $pRqs$ buffer. We will now prove that $dirw[addr(r')][child(r')]$ cannot be set once $r$ has been enqueued into the $pRqs$ buffer.

The directory-wait state can be set only by **Rule**⟨*DwnRqEvict*⟩, **Rule**⟨*DwnRq*⟩ or **Rule**⟨*DwnRqP*⟩. **Rule**⟨*DwnRqEvict*⟩ cannot fire since it requires the presence of a ⟨WaitV, $l$⟩-request where slot $l$ contains $addr(r)$. But there is already a ⟨WaitSt, _⟩-request for $addr(r)$ which must point to the same slot because of Invariant 15, thus violating Invariant 12. **Rule**⟨*DwnRq*⟩ cannot fire since it never sends a downgrade request to the child which sent the request. Finally, **Rule**⟨*DwnRqP*⟩ cannot fire since it explicitly checks that there is no ⟨WaitSt, _⟩-request whose $toState$ is less than or equal to the corresponding cache state. □

## 8.4 Conclusion

In this chapter we verified the directory-based cache-coherence protocol implemented over an arbitrary hierarchy of caches. We proved that this complex system refines a much simpler atomic memory specification. We had to use several properties about refinements discussed in Section 6.5 in this proof. But the bulk of the proof is in identifying all the invariants that the complex system obeys. This chapter lists all these invariants and a proof sketch for these invariants.

# Chapter 9

# Conclusions and Future Work

In this thesis we developed a framework for verifying hardware systems expressed in Bluespec using the theory of labeled transition systems and implemented this framework using the Coq proof assistant. Our formalization of a system implementing a specification is based on *trace refines* relation. Our methodology enables modular verification as we map the semantics of Bluespec programs with interfaces (*i.e.*, method calls or definitions) to LTSes with the labels of the transitions representing the communication that happens in and out of a Bluespec module.

Using this framework we modularly verified that a complex multiprocessor consisting of speculative out-of-order cores connected to a coherent cache hierarchy implements the sequential consistency specification. We first verified that a speculative out-of-order core implements a simple decoupled processor where the requests to memory are decoupled from responses to memory. We then verified that the coherent cache hierarchy implements a simple atomic memory specification which has buffers around an instantaneous memory into which requests and response from and to the processors reside. Finally, we proved that the simple decoupled processor implements an instantaneous processor where the requests and responses are not decoupled. The system consisting of instantaneous processors and the instantaneous memory is the

textbook definition of sequential consistency, thus completing the proof.

The cache-coherence protocol that is employed in our multiprocessor system is a realistic directory-based protocol which supports arbitrary tree hierarchies of caches. It is fairly complex, and much of the verification effort was in proving that the cache-coherence protocol implements the atomic memory.

While this thesis has covered the verification aspect of programs written in Bluespec thoroughly, this work is orthogonal to the issue of synthesis of Bluespec programs and generating hardware circuits from it. While there is a commercial compiler available for the said task, it is not formally verified. Ultimately, we want a completely verified circuit at least before fabrication. The first step towards this goal is to verify that the hardware circuits generated post synthesis also match the semantics of the Bluespec program. Proving that scheduling of multiple rules concurrently is correct is the first step in any such compilation/synthesis verification. Some initial work in this area has been done by Braibant *et al.* [13] for single-module Bluespec programs without any external method calls. In order to prove properties about scheduling, we believe that the notion of *weak-implements* relation discussed in Section 6.6 is important, as the label sequence in a scheduled program will have merged labels compared to the original program.

While the foundations for a hardware verification framework have been designed and established during the course of this thesis, there is still some work left to enable widespread use of our methodology for hardware verification. The main area that could use further extensions is proof automation. As discussed in Section 5.2, the Coq proof assistant provides a lot of assistance for proof automation via the Ltac tactic language. A heavy use of these techniques will nevertheless reduce the human effort needed to verify complex systems, by discharging a lot of proof obligations automatically. Another important requirement for the widespread use of this methodology is the creation of an extensive library of commonly used hardware components that are

already verified. Practically, as can be seen in other software programming languages, the presence of a good library is one of the main factors for widespread adoption.

# Appendix A

# Functions Used in Defining Semantics

## A.1 Extracting Values from Finite Register Maps

We now define the operation $o(r)$ which is used to extract the value of register $r$ from the finite map $o$.

**Definition 47.**

$$o(r) \triangleq \begin{cases} \epsilon & : \quad o = [] \\ v & : \quad o = (r \mapsto v) :: o' \\ o'(r) & : \quad o = (x \mapsto v) :: o', x \neq r \end{cases}$$

# Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] Robert Alverson, Duncan Roweth, and Larry Kaplan. The gemini system interconnect. In *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, HOTI '10, pages 83–87, Washington, DC, USA, 2010. IEEE Computer Society.

[3] Arvind and James C. Hoe. Digital Circuit Synthesis System. United States Patent US 6,597,664 B1, July 2003.

[4] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave. High-level Synthesis: An Essential Ingredient for Designing Complex ASICs. In *Proceedings of ICCAD'04*, San Jose, CA, 2004.

[5] Arvind and Xiaowei Shen. Using term rewriting systems to design and verify processors. *Micro, IEEE*, 19(3):36–46, 1999.

[6] Lennart Augustsson, Jacob Schwarz, and Rishiyur S. Nikhil. Bluespec Language definition, 2001. Sandburst Corp.

[7] R. J. R. Back and J. von Wright. Trace refinement of action systems. In *Structured Programming*, pages 367–384. Springer-Verlag, 1994.

[8] Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar. Multi-core cache hierarchies. *Synthesis Lectures on Computer Architecture*, 6(3):1–153, 2011.

[9] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[10] Ritwik Bhattacharya, Steven German, and Ganesh Gopalakrishnan. Symbolic partial order reduction for rule based transition systems. In Dominique Borrione and Wolfgang Paul, editors, *Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 332–335. Springer Berlin Heidelberg, 2005.

[11] Ritwik Bhattacharya, Steven M. German, and Ganesh Gopalakrishnan. Exploiting symmetry and transactions for partial order reduction of rule based specifications. In *Antti Valmari, editor, SPIN, volume 3925 of Lecture Notes in Computer Science*, pages 252–270. Springer, 2006.

[12] Bluespec, Inc., Waltham, MA. *Bluespec SystemVerilog Version 3.8 Reference Guide*, November 2004.

[13] Thomas Braibant and Adam Chlipala. Formal verification of hardware synthesis. In *CAV 2013, 25th International Conference on Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2013.

[14] StephenD. Brookes and WilliamC. Rounds. Behavioural equivalence relations induced by programming logics. In Josep Diaz, editor, *Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*, pages 97–108. Springer Berlin Heidelberg, 1983.

[15] Jerry R Burch and David L Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification*, pages 68–80. Springer, 1994.

[16] H.W. Cain and M.H. Lipasti. Memory ordering: a value-based approach. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 90–101, June 2004.

[17] P Camurati, F Corno, P Prinetto, C Bayol, and B Soulas. Inter-process communications for system-level design. In *at International Workshop on Hardware/Software Codesign, Cambridge*, 1993.

[18] Paolo Camurati, Fulvio Corno, and Paolo Prinetto. A methodology for system-level design for verifiability. In *Correct Hardware Design and Verification Methods*, pages 80–91. Springer, 1993.

[19] Paolo Camurati, Fulvio Corno, Paolo Prinetto, Catherine Bayol, and Bernard Soulas. System-level modeling and verification: a comprehensive design methodology. In *European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings.*, pages 636–640. IEEE, 1994.

[20] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: A performance view. *IBM J. Res. Dev.*, 51(5):559–572, September 2007.

[21] Xiaofang Chen, Yu Yang, Ganesh Gopalakrishnan, and Ching-Tsun Chou. Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. *Form. Methods Syst. Des.*, 36(1):37–64, February 2010.

[22] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. ACM.

[23] Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the coq proof assistant.* MIT Press, 2013.

[24] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer Aided Design*, pages 382–398. Springer, 2004.

[25] Paul Curzon and IM Leslie. A case study on design for provability. In *Engineering of Complex Computer Systems, 1995. Held jointly with 5th CSESAW, 3rd IEEE RTAW and 20th IFAC/IFIP WRTP, Proceedings., First IEEE International Conference on*, pages 59–62. IEEE, 1995.

[26] Ed Czeck, Ravi Nanavati, and Joe Stoy. Reliable Design with Multiple Clock Domains. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, 2006.

[27] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[28] WilliamJ. Dally and CharlesL. Seitz. The torus routing chip. *Distributed Computing*, 1(4):187–196, 1986.

[29] W.J. Dally. Express cubes: improving the performance of k-ary n -cube interconnection networks. *Computers, IEEE Transactions on*, 40(9):1016–1023, Sep 1991.

[30] Nirav Dave. Designing a Reorder Buffer in Bluespec. In *Proceedings of MEMOCODE'04*, San Diego, CA, 2004.

[31] Nirav Dave, Arvind, and Michael Pellauer. Scheduling as Rule Composition. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Nice, France, 2007.

[32] Nirav Dave, Man Cheuk Ng, and Arvind. Automatic Synthesis of Cache-Coherence Protocol Processors Using Bluespec. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Verona, Italy, 2005.

[33] Nirav Dave, Michael Pellauer, Steve Gerding, and Arvind. 802.11a Transmitter: A Case Study in Microarchitectural Exploration. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Napa, CA, 2006.

[34] David Delahaye. A Tactic Language for the System Coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Computer Science (LNCS)/Lecture*

*Notes in Artificial Intelligence (LNAI)*, pages 85–95, Reunion Island (France), November 2000. Springer.

[35] Giorgio Delzanno. Automatic verification of parameterized cache coherence protocols. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer Berlin Heidelberg, 2000.

[36] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *Computer Design: VLSI in Computers and Processors, 1992. ICCD '92. Proceedings, IEEE 1992 International Conference on*, pages 522–525, Oct 1992.

[37] E. Allen Emerson and Vineet Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, pages 247–262, 2003.

[38] D.B. Glasco, B.A. Delagi, and M. Flynn. Update-based cache coherence protocols for scalable shared-memory multiprocessors. In *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, volume 1, pages 534–545, Jan 1994.

[39] B. Grot, J. Hestness, S.W. Keckler, and O. Mutlu. Express cube topologies for on-chip interconnects. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 163–174, Feb 2009.

[40] P. Hammarlund, A.J. Martinez, A.A. Bajwa, D.L. Hill, E. Hallnor, Hong Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R.B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The fourth-generation intel core processor. *Micro, IEEE*, 34(2):6–20, Mar 2014.

[41] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[42] 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, pages 289–300, New York, NY, USA, 1993. ACM.

[43] Charles Antony Richard Hoare. *Proof of correctness of data representations.* Springer, 2002.

[44] James C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions. In *Proceedings of ICCAD'00*, pages 511–518, San Jose, CA, 2000.

[45] James C. Hoe and Arvind. Operation-Centric Hardware Description and Synthesis. *IEEE TRANSACTIONS on Computer-Aided Design of Integrated Circuits and Systems*, 23(9), September 2004.

[46] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. A 5-ghz mesh interconnect for a teraflops processor. *Micro, IEEE*, 27(5):51–61, Sept 2007.

[47] Chung-Wah Norris Ip, David L. Dill, and John C. Mitchell. State reduction methods for automatic formal verification, 1996.

[48] Ranjit Jhala and Kenneth L. McMillan. Microarchitecture verification by compositional model checking. In *CAV*, pages 396–410. Springer-Verlag, 2001.

[49] He Jifeng. Process simulation and refinement. *Formal Aspects of Computing*, 1(1):229–241, 1989.

[50] Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark R. Tuttle, and Yuan Yu. Checking cache-coherence protocols with TLA$^+$. *Formal Methods in System Design*, 22(2):125–131, 2003.

[51] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, et al. Replacing testing with formal verification in Intel® Core$^{tm}$ i7 processor execution engine validation. In *Computer Aided Verification*, pages 414–429. Springer, 2009.

[52] Michal Karczmarek and Arvind. Synthesis from multi-cycle atomic actions as a solution to the timing closure problem. In *ICCAD*, 2008.

[53] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[54] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the window of vulnerability in multiphase memory transactions. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 274–284, New York, NY, USA, 1992. ACM.

[55] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Apr 1994.

[56] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[57] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[58] Butler W. Lampson. Principles of computer systems.

[59] Panagiotis Manolios and Sudarshan K. Srinivasan. S.K.: Automatic verification of safety and liveness for pipelined machines using WEB refinement. *ACM Trans. Des. Autom. Electron. Syst*, pages 1–19, 2008.

[60] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *CHARME 01: IFIP Working Conference on Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science 2144*, pages 179–195. Springer, 2001.

[61] Kenneth L McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Computer Aided Verification*, pages 110–121. Springer, 1998.

[62] K.L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 219–237. Springer Berlin Heidelberg, 1999.

[63] K.L. McMillan and James Schwalbe. Formal verification of the Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 111–134, 1992.

[64] George J Milne. Design for verifiability. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 1–13. Springer, 1990.

[65] Robin Milner. The polyadic pi-calculus: a tutorial. Technical report, Logic and Algebra of Specification, 1991.

[66] J Strother Moore. An ACL2 proof of write invalidate cache coherence. In *Proc. CAV'98, volume 1427 of LNCS*, pages 29–38. Springer, 1998.

[67] Man Cheuk Ng, Muralidaran Vijayaraghavan, Gopal Raghavan, Nirav Dave, Jamey Hicks, and Arvind. From WiFI to WiMAX: Techniques for IP Reuse Across Different OFDM Protocols. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Nice, France, 2007.

[68] Grace Nordin and James C. Hoe. Synchronous Extensions to Operation-Centric Hardware Description Languages. In *Proceedings of MEMOCODE'04*, San Diego, CA, 2004.

[69] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 348–354, New York, NY, USA, 1984. ACM.

[70] I.E. Papazian, S. Kottapalli, J. Baxter, J. Chamberlain, G. Vedaraman, and B. Morris. Ivy bridge server: A converged design. *Micro, IEEE*, 35(2):16–25, Mar 2015.

[71] Seungjoon Park and David L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296. ACM Press, 1996.

[72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[73] Juan C. Pichel and Francisco F. Rivera. Sparse matrixâĂŞvector multiplication on the single-chip cloud computer many-core processor. *Journal of Parallel and Distributed Computing*, 73(12):1539 – 1550, 2013. Heterogeneity in Parallel and Distributed Computing.

[74] John C Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6(3-4):233–247, 1993.

[75] Daniel L. Rosenband. The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs. In *Proceedings of MEMOCODE'04*, San Diego, CA, 2004.

[76] Daniel L. Rosenband and Arvind. Modular Scheduling of Guarded Atomic Actions. In *Proceedings of DAC'04*, San Diego, CA, 2004.

[77] Daniel L. Rosenband and Arvind. Hardware Synthesis from Guarded Atomic Actions with Performance Specifications. In *Proceedings of ICCAD'05*, San Jose, CA, 2005.

[78] C. Scheurich and M. Dubois. The design of a lockup-free cache for high-performance multiprocessors. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pages 352–359, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[79] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[80] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-reconcile & fences (CRF): a new memory model for architects and compiler writers. In *Proceedings of the 26th annual international symposium on Computer architecture*, pages 150–161. IEEE Computer Society, 1999.

[81] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[82] Per Stenström, Fredrik Dahlgren, and Lars Lundberg. A lockup-free multiprocessor cache design. In *ICPP (1)*, pages 246–250, 1991.

[83] M. Talupur and Mark R. Tuttle. Going with the flow: Parameterized verification using message flows. In *Formal Methods in Computer-Aided Design, 2008. FMCAD '08*, pages 1–8, Nov 2008.

[84] M.B. Taylor, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, and J. Kim. Evaluation of the raw microprocessor: an exposed-wire-delay architecture for ilp and streams. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 2–13, June 2004.

[85] Nuutti Varis and Jukka Manner. In the network: Sandy bridge versus nehalem. *SIGMETRICS Perform. Eval. Rev.*, 39(2):53–55, September 2011.

[86] Muralidaran Vijayaraghavan, Nirav Dave, and Arvind. Modular compilation of guarded atomic actions. In *11th ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMCODE 2013, Portland, OR, USA, October 18-20, 2013*, pages 177–188, 2013.

[87] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[88] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, September 2007.

[89] Phillip J. Windley. Formal modeling and verification of microprocessors. *Computers, IEEE Transactions on*, 44(1):54–72, 1995.

[90] Meng Zhang, Jesse D. Bingham, John Erickson, and Daniel J. Sorin. Pvcoherence: Designing flat coherence protocols for scalable verification. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 392–403. IEEE Computer Society, 2014.

[91] Meng Zhang, Alvin R. Lebeck, and Daniel J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 471–482, Washington, DC, USA, 2010. IEEE Computer Society.