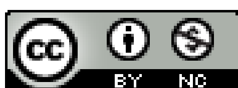
SPECIFYING AND VALIDATING MEMORY PROTOCOLS FOR CHIP GENERATORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Megan A. Wachs
June 2013

© 2013 by Megan Anneke Wachs. All Rights Reserved.
Re-distributed by Stanford University under license with the author.



This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.
http://creativecommons.org/licenses/by-nc/3.0/

This dissertation is online at: http://purl.stanford.edu/wg083np6594

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christoforos Kozyrakis

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Subhasish Mitra

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Shared memory protocols are a complex and active area of research. A memory protocol describes the way in which multiple processing entities communicate to achieve a consistent view of a shared memory system. From a research perspective, it is easiest to specify a protocol at a high level, in order to emphasize what is new and interesting about the protocol. With only this specification it is a challenge to evaluate these solutions in hardware, because creating an entire hardware chip multiprocessor platform is a complex and error prone task. This thesis presents Specification Language for Advanced Memory Models (SLAMM), a language for specifying protocols at a high level in a C-like language. A SLAMM specification is compiled to configure a hardware template which can implement the memory protocol. This overall system template is composed of a number of independent controllers, and we present in detail the template for one such controller and the microblocks which compose it. We also describe a technique for verifying the resulting hardware system using a Relaxed Scoreboard, which allows either random testing of the system or monitored directed tests. These parts taken together provide a flexible memory platform that is easy to program and validate to evaluate new protocols in hardware.

Acknowledgements

This thesis would not exist without the help and support of the many wonderful people I had the good fortune to know or meet during my years at Stanford. First and foremost, I thank Professor Mark Horowitz, for his always spot-on guidance, patience, and trust. It was a privilege to be taught by him and to join his research group, and I always left our discussions feeling that anything was possible. I depart from Stanford having learned to show things are great rather than saying so, and to always wonder, "What's new and interesting in the world?"

Professor Subhasish Mitra was generous enough to serve as my co-advisor. He provided his expert knowledge and great advice, always accompanied by a smile and genuine interest. To Christos Kozyrakis, I extend my gratitude for teaching me much of what I know about processors in my first years at Stanford, for his feedback and support on my research as I was starting out, and for serving on my thesis committee. I am grateful to Professor David Mazieres for acting as Chair at my thesis defense, but also for providing me with entertaining lectures and the chance to TA his class. To Teresa Lynn and Mary Jane Swenson, I extend a special thanks for everything they do to make grad student research run smoothly.

I was lucky enough to be supported by Sequoia Capital Stanford Graduate Fellow during my first years at Stanford, allowing me to explore my research options, and for providing continuing connections to this day.

I remember the first day of Stanford EE orientation, being intrigued by all the accents asking questions. I am grateful to have such amazing peers from around the world. First and foremost to Ofer Shacham, I extend my great thanks for being a wonderful collaborator and friend, and for always maintaining an inspring and positive attitude. I already miss the good times in Gates 320. I thank Amin Firoozshahian and Alex Solomatnikov for allowing me to crash in their office and join the Smart Memories project, and for their mentorship, lessons on architecture, and energetic debates. Many thanks to Kyle Kelley,

Andrew Danowitz, Omid Azizi, and the rest of the VLSI research group, for all the excellent collaborations, research advice, lunch conversations, and snow cabin adventures. Thanks also to the Pervasive Parallelism Lab and all its members for the feedback on my work and interesting discussions over the years. A special thanks to Steve Richardson for his always good-natured mentorship, encouragement, and listening ear.

My research was sustainable only due to the friends and teammates that kept me balanced and sane outside of Gates. Meg Desko literally opened the door for me my first day on Stanford and provided opportunities for venting, cookies, and encouragement in the years that followed. Thanks to Chris Tschinkel for our many adventures and his absolute faith in me. To everyone from Glenwwod Junction, the Brown Mafia, the Triathlon Team, the Dolphin Club, and the rest of my friends old and new, I thank you for making my life so enjoyable, and helping me get outside to explore the West Coast. Special thanks to Sameh Galal for starting thesis boot camp and to Oliver Hickman for teaching me about the fifty minute session, without which this thesis would never have come together.

My greatest appreciation is for the encouragement from my engineer-laden extended family, who taught me to get excited about building things and solving problems. My brother Jason and sister Laura never let me take myself too seriously. Thanks to my father Steve Wachs, for always including me in his projects and for believing in me without fail. Lastly, to my mother, Barbara Wachs, thank you for your love and support, and for making it seem completely routine to be a woman in computing.

Contents

Abstract **v**

Acknowledgements **vi**

1 Introduction **1**

2 Programmable Memory System Concepts **3**

2.1 Programmable Memory System Platforms 4

2.2 A Platform for Memory Protocol Experimentation: Stanford Smart Memories 6

2.3 Programming Limitations of Smart Memories 13

2.4 The Generator Approach 16

2.4.1 Building a Generator for a Memory System 18

2.4.2 Protocol Specification Levels 18

3 A Programmable Hardware Architecture **23**

3.1 A Hierarchical Memory System 23

3.2 Describing a Memory Protocol 27

3.3 High-Level System Template 30

3.4 Generalized Protocol Controller 30

3.4.1 Interfaces 32

3.4.2 Events and Transitions 35

3.4.3 Translations 38

3.4.4 State Tracking and Special Blocks 45

4 Programming the Hardware: SLAMM **53**

4.1 The SLAMM Constructs 54

4.1.1 Messages . . . 54
4.1.2 State and State Updates . . . 59
4.1.3 Control Flow . . . 60
4.1.4 Special Blocks . . . 63
4.2 Integrating SLAMM Parameters . . . 66
4.2.1 The SLAMM Compiler . . . 68
4.3 Mapping SLAMM Constructs to the Architectural Model . . . 70
4.3.1 Interfaces Between Blocks . . . 70
4.3.2 Protocol Controller . . . 71
4.3.3 Memory Block . . . 72
4.3.4 Reply Handler . . . 73
4.3.5 Off-Chip Memory . . . 73
4.4 Extending a Protocol with SLAMM . . . 73

5 Relaxed Scoreboard 75
5.1 Problem Definition . . . 77
5.2 Relaxed Scoreboard . . . 80
5.2.1 Write Atomicity . . . 84
5.2.2 Transaction Isolation in TCC Memory Model . . . 85
5.2.3 Store Ordering in a TSO Memory System . . . 86
5.3 Evaluation . . . 88
5.3.1 Scoreboard Design for Smart Memories . . . 88
5.3.2 Quality of Results . . . 89
5.3.3 Performance and Overheads . . . 90
5.4 Relaxed Scoreboard Conclusions . . . 93

6 Conclusions & Future Work 95

A Relaxed Scoreboard Details 101
A.1 Introduction . . . 101
A.2 Internal Structure of The Relaxed Scoreboard . . . 101
A.3 Relaxed Scoreboard Classes Summary . . . 102
A.3.1 Trace_trans Class . . . 103
A.3.2 Trace_trans_Q Class . . . 104

A.3.3 TCC_transaction Class 105
A.3.4 Scoreboard Class 106
A.4 Interfacing with The Relaxed Scoreboard 107
A.5 Relaxed Scoreboard Operation 108
A.5.1 Address Translation Step 109
A.5.2 The CHECK Step 110
A.5.3 The UPDATE Step 112
A.5.4 Additional Scoreboard Functions 124
A.5.5 Scoreboard TCC Tasks 125
A.6 Garbage Collection 131
A.7 Scoreboard Output and Error Reporting 132

B Full SLAMM Grammar **133**

Bibliography **137**

List of Tables

4.1 SLAMM Constructs 64

5.1 Errors Found by the Relaxed Scoreboard 91
5.2 Checks and Updates for Memory Protocols 92

List of Figures

2.1 LUT-Based FSM 4
2.2 Stanford Smart Memories Architecture 6
2.3 Smart Memories Protocol Controller 7
2.4 SSM's Configurable Local Memory 8
2.5 SSM Memory Mat PLA 9
2.6 Memory Mat Configuration Spreadsheet 10
2.7 Smart Memories Configuration Flow 11
2.8 Smart Memories Protocol Controller Code 12
2.9 Smart Memories Die Photo 14
2.10 Using a Chip Generator 17
2.11 Memory System Specification Levels 19

3.1 Hierarchical Memory System Architecture 24
3.2 CMP Memory System Abstraction 25
3.3 Instance of a CMP Template 26
3.4 A Simple Memory Protocol 28
3.5 Generalized Protocol Controller 31
3.6 Hardware Interface Primitive 32
3.7 Uses of the Interface Primitive 33
3.8 Event Trigger TCAM 38
3.9 Transition TCAM 39
3.10 Action to Message Translations 42
3.11 Translation Hardware 43
3.12 State Tracking Hardware 45
3.13 Two Dimensional State Tracking Hardware 47

3.14 Message Pipe Hardware 49
3.15 Control Signal Generation Hardware 50

4.1 An Example SLAMM Specified Flow 55
4.2 State Machine Code Example 56
4.3 Message Code Examples 57
4.4 State Enumeration Code Examples 59
4.5 Control Flow Code Examples 61
4.6 Special Block Code Example 63
4.7 Summarized SLAMM Grammar 65
4.8 Genesis I/O Loop 66
4.9 SLAMM Compiler Flow 67

5.1 Non-determinism of a CMP Memory System. 81
5.2 Write atomicity example. 83
5.3 Total Store Order Example. 87
5.4 Relaxed Scoreboard Uncertainty Window 90

Chapter 1

Introduction

The notion of using caches, small fast memory close to the processor, to hide memory latency is an old idea [48]. Cache complexity grows in a multiprocessor context, because for processors to work concurrently on a shared task, the data that they read and write locally must be, in general, visible to other processors. This requires rules about how and when the changes that they make should be observable by other processors. This problem is called coherency and consistency, and a good deal of research havs been devoted to developing shared memory protocols that can efficiently provide easily understandable coherency and consistency models, much of it in the 1980's [32, 5]. While these are well established problems with many excellent solutions, researchers are still innovating the hardware, for a variety of reasons. For example, Sanchez et al proposed ZCache [34], a way to find a larger pool of good eviction candidates without slowing down the common case cache lookups. ZCache helps the miss rate of caches and therefore has direct benefits for many applications. Wang et al proposed PLCache [47], secure hardware that is immune to certain timing attacks by introducing different requirements for selecting eviction candidates. In such cases, performance is not the issue as much as isolating different processes from each other at a hardware level without stymying performance. A final example, Cheriton et al proposed HICAMP [11], which provides a different view of a memory system. Rather than the traditional memory model of writing data to known addresses in the system, a HICAMP "write" requests the address of known data. A read is still achieved by looking up the data at a known address. This technique makes concurrency easier because the data at an address is immutable.

As researchers innovate and propose new protocols and techniques, they often create

hardware simulators to evaluate their ideas, but eventually want to verify how these ideas would perform on real hardware. An entire CMP system is exceedingly complex to design and verify, but only relatively small changes should be needed to evaluate a new memory protocol. The Stanford Smart Memories project aimed to evaluate this idea, by creating a single flexible hardware platform to support multiple memory protocols. It was able to show that the similarities between memory protocols were greater than their differences, and was able to support multiple protocols that were active areas of research. However, despite its design for flexibility, the hardware that was taped out was not flexible enough to implement ideas not considered during the design phase. The Smart Memories platform also provided no higher level interface for easy programming of the memory protocol, which made it difficult if not impossible for a researcher with a new protocol in mind to correctly configure the hardware to implement it.

This thesis addresses the issue of how to easily specify a desired protocol in a higher level language, and then have that protocol converted into synthesizable hardware. To accomplish this task, we create a flexible template and a compiler that can take our protocol description and generate the parameters and configuration files the template needs to use.

Chapter 2 reviews the solutions used for creating and programming hierarchical memory protocols in earlier projects. We identify the need for an easier way to specify protocols which can be implemented in hardware. Chapter 3 first describes the abstract architecture of a modern hierarchical memory system, and describes how it is composed of smaller hardware templates which are able to take configuration parameters to shape their behavior. Since each of these smaller templates must be configured as part of a whole, Chapter 4 describes the high-level language used to specify a memory protocol and how it generates the parameters needed for the templates. Finally, we know from experience that creating an implementation is not enough, it must also be verified, so Chapter 5 describes a technique for verifying implementations of shared memory protocols at a high level. In Chapter 6 we examine the utility of this approach and point out possible avenues for future work.

Chapter 2

Programmable Memory System Concepts

Academia and industry have aggressively moved towards Chip Multiprocessors (CMP) as the main processing unit in current and future compute platforms. A major challenge in building these complex systems is determining how the multiple processors should communicate and effectively use shared memory resources. This debate regarding the "right" programming model(s) for these machines, and hence how the memory system should be implemented, is far from being settled. While some advocate streams [16, 23, 31] due to their high compute density per Watt, streams have a more limited application space. Others prefer Thread Level Speculation [17, 42], recently generalized and formulated into the Transactional Memory model [26,18], for its broader application domain and ease of use from the programmer's perspective. Meanwhile, the most prevalent general purpose platforms are still variations of a cache coherent multi-thread model [3,30,22]. Several current research directions build on this powerful model, by enhancing the tag mapping and replacement policies. For example, while maintaining the overall cache coherent multi-thread model, by modifying some lower level aspects, researchers are pursuing greater performance [34], enhanced security [47, 9], and efficient snapshotting of the entire memory [11].

Historically, as designers explored different ways of creating these shared memory systems, they composed more complex systems in hardware to evaluate their ideas. In order to create a system to implement a new memory protocol, the designers often built in a level of programability, either for ease of implementation or the ability to make adjustments and corrections after the system was constructed. Programmability also extends the

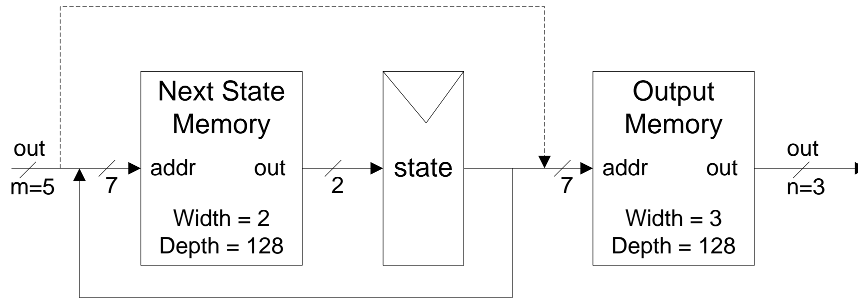


Figure 2.1: An implementation of a Finite State Machine implemented using lookup tables.

utility of a system by allowing it to scale, because components can be configured to create larger networks when the routing and communication between them is configurable. The following section reviews some of these systems to consider why and how they were made programmable. As the ability to put more complexity on die increased, researchers realized that this programmability could be designed into the hardware as a first order design goal to enable the evaluation of new ideas on a single platform. Section 2.2 describes a system which takes these ideas to the extreme and is completely configurable in hardware, the Stanford Smart Memories project. In Section 2.3 we describe the limitations of even this degree of flexibility in hardware, and the challenges encountered in programming it. Section 2.4 proposes using a *generator* approach to achieve an efficient implementation in hardware without the limitations encountered in even configurable hardware. Since the challenge of programming the hardware still exists, in Section 2.4 we also describe some approaches to describing memory protocols that makes them easier to implement and verify, techniques that we will build on in this thesis to create a flexible memory system that is easy to configure.

2.1 Programmable Memory System Platforms

In earlier multiprocessor systems, programmability was built into the memory system not from a desire for the system to be configurable per se, but to allow debugging and corrections in a pre- and post-silicon system. A general technique for implementing programmable state machines is to use microcode memories. Essentially, a lookup table (LUT) is used to generate control signals and next state logic, as shown in Figure 2.1. The same LUT structure can be used to create microcodes for a system. Many memory systems share

similar structures and primitives.

The MIT Alewife machine [4] consisted of Sparcle processors connected by a Communications and Memory Management Unit (CMMU). The CMMU was a standalone chip that could handle much of the low-level primitives needed for coherent shared memory, but could trap into software for handling more complicated sharing situations. This was done to make the common case fast and efficient without introducing a lot of specialized hardware for less common, more computationally intense scenarios.

The SGI Origin [27] processor used a table-based technique to implement part of its scalable cache coherent system. The system consisted of several processors locally connected by *Hub* chips which communicated over a scalable interconnection network. Each Hub core's behavior was dictated by the code in its *Protocol Table*, a microcode memory of the sort discussed above used to implement the cache protocol. To avoid the overheads introduced by using a reprogrammable look up table, the Protocol Tables were hardcoded in the final hardware. More recent work by K. Kelley, the author, et al [24] has shown that with sufficient annotation, a table based description can be synthesized into an efficient gate implementation. The challenge of using a LUT/microcode memory becomes that of determining the appropriate values for the LUT, and providing annotation information. The protocols encoded the SGI Origin processor's tables were verified using formal techniques.

The Stanford FLASH project had a flexible memory architecture supporting both cache coherency and message passing, and had a custom node controller (the MAGIC chip) which consisted of both hardwired data paths and a programmable processor specialized for executing protocol operations [20]. The use of the protocol processor simplified the design and implementation of the system, and made it more flexible. The programmable processor implemented a subset of the DLX instruction set [21]. However, it added special extensions for bitwise operations and sending messages to the special hardware units which could perform the basic protocol operations. The protocol processor design was simple, and much of the burden of ensuring deadlock avoidance and protocol correctness was the responsibility of the protocol programmer. This programmability allowed experimentation and correction of the most difficult parts of the protocol after the hardware had taped out.

The fact that the same primitive operations could be used to implement a variety of shared memory protocol configurations did not go unnoticed. Rather than building a system which could be coded to implement a set coding (like the SGI Origin) or a system which could execute code, the Stanford Smart Memories project fleshed out the idea of using

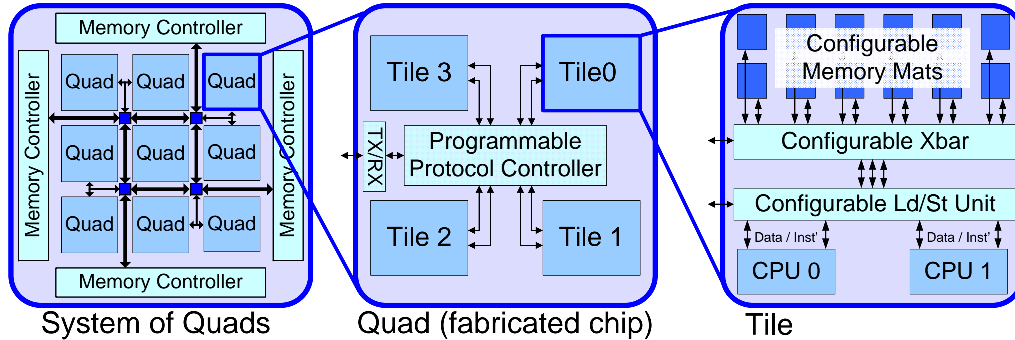


Figure 2.2: Stanford Smart Memories Reconfigurable Architecture. Reconfigurable blocks are shown as lighter.

flexible hardware primitives to implement a variety of protocols in hardware. The following section describes the Smart Memories system in more detail.

2.2 A Platform for Memory Protocol Experimentation: Stanford Smart Memories

Stanford Smart Memories was a research project which extended the idea of programmable hardware to build a single hardware platform that could support multiple programming models by creating a flexible execution and memory system. Its goal was to show that, from a hardware perspective, the similarities between many programming models are greater than the differences. The project initially aimed to meet this goal by creating a configurable data-path for the processor core, and adding programmability to the memory system [28]. This conceptual design leveraged the fact that all memory models rely on physical memory storage in proximity to the processing unit, and on controllers that orchestrate data movements among these repositories and to and from the main memory. In addition, most memory systems need some “state” to be associated with the data (e.g., valid bit in caches or speculatively read/write bits in Transactional Memory), and therefore the Smart Memories memory system added meta-data bits to its local storage arrays. The differences between memory models were created by defining the meaning of the meta-data bits, and by defining the protocols for the actions that need to be taken when specific conditions occur.

The resulting Smart Memories (SM) eight-core CMP presented here is a fully implemented system: from an architectural simulator written in C++, through complete

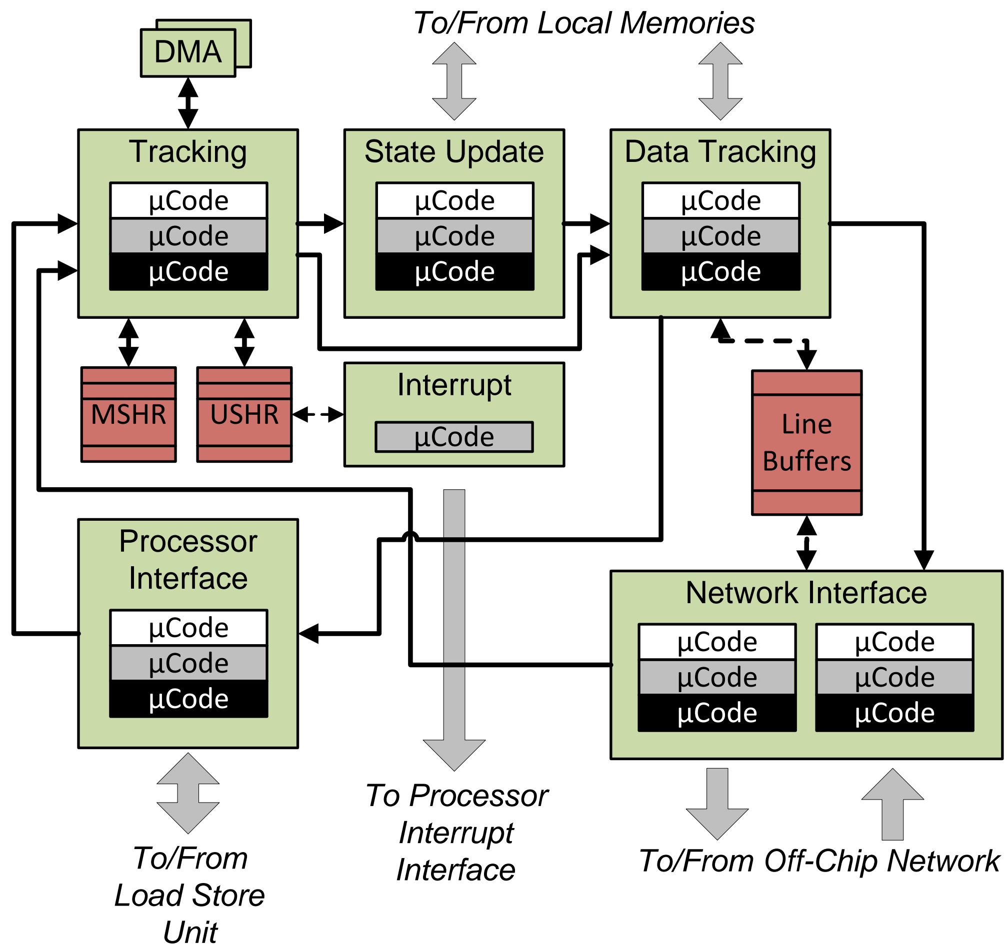
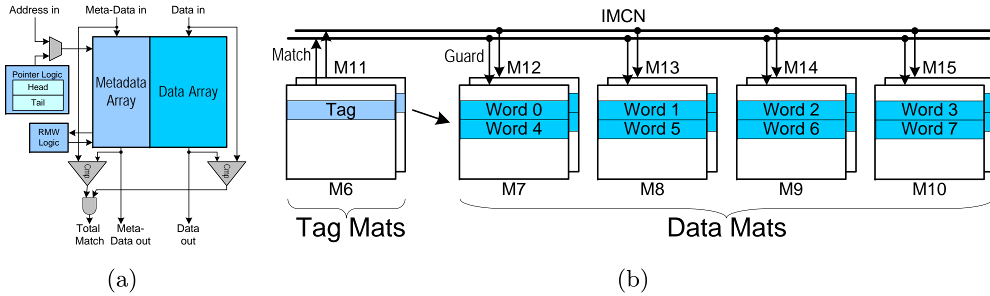


Figure 2.3: The Smart Memories Protocol Controller. The controller had several modules with microcode-driven sections with configurable behavior and message types. The interfaces between the blocks were fixed.



(a) (b)

Figure 2.4: Configurable local memory. (a) Block diagram of the memory mat. (b) Example mat organization for a 2-way cache.

RTL/gate simulation environment, to the silicon chip that fabricated in a 90*nm* technology at STMicroelectronics [1]. Four SM chips have been integrated together in a system instrumented with additional FPGAs to provide full 32-core functionality. The software layer includes a C/C++ compiler[1] and runtime environments for the stream, transactional coherence and consistency (TCC) [18] and multi-thread execution models.

Smart Memories is a hierarchical system, as shown in Figure 2.2. Each individual chip, or *quad*, consists of a set of four *tiles*, where each tile contains two processors and a set of *memory mats*. The tiles communicate off-chip and with each other via a *protocol controller*. The protocol controller can make requests to an off-chip *memory controller*, but can also communicate through a generic network to other protocol controllers, to build a truly hierarchical system. Each of these modules contain configurable, programmable features which enable them to implement different memory protocols. This section describes these features, and also highlights the techniques used generate the correct memory programming configuration for each system.

The memory mats contained many programmable components, which shaped the functionality of each mat and the way that they were connected together. While most of the mat configuration was manageable enough to be determined by hand, one aspect required good higher-level programming support. This was programming the behavior of the meta data bits in the mat. Each address in a memory mat had a few meta-data bits whose behavior was programmable. These bits were controlled by a separate opcode from the main data bits at each address, so they could be read or written directly in parallel with data operations. Additionally, they could be accessed with a special read-modify-write operation. The

[1] Tensilica compiler instrumented with special Smart Memories TIE instructions [15].

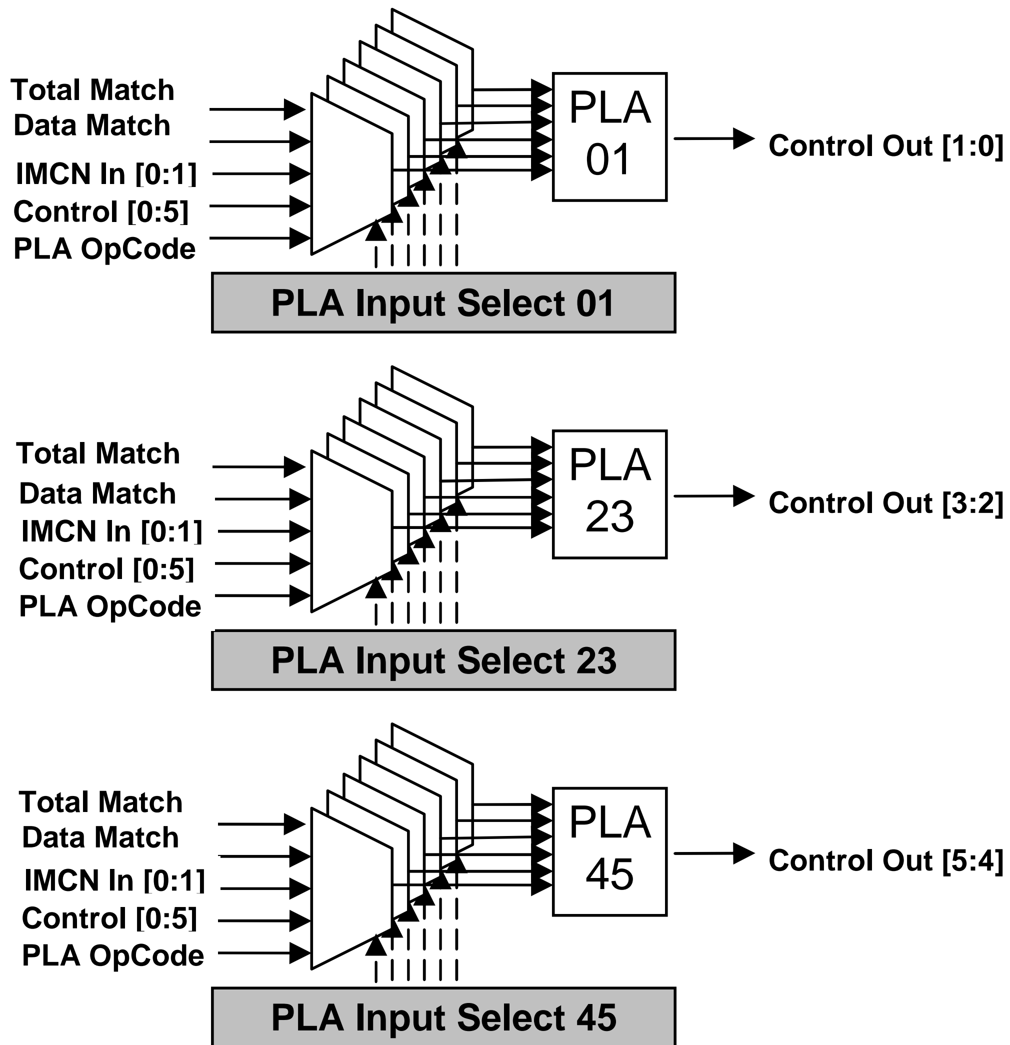


Figure 2.5: SSM Memory Mat PLA. Each Programmable Logic Array was itself programmable to output 2 of the 6 total output meta-data bits.The inputs were also selected by another configuration register, such that 6 inputs provided the PLA address, out of a possible set of 13 inputs.

Figure 2.6: Memory Mat PLA Configuration Spreadsheet Screenshot. The blocks were color coded to guide the user into what they needed to fill out and copy into the configuration file.

action of the modify operation was programmable. For example, a memory mat on a cache load could conditionally set the 'M' (modified) state bit if and only if there was a hit and and this was the hitting way, while other ways would clear the 'M' bit on a hit. The logic for this behavior for each of the configurable bits (and separately for mats which held tag bits and those which held data bits) was specified in a large lookup table, or Programmable Logic Array (PLA). In order to keep this lookup table to a reasonable size, there was a preselection configured for what inputs would actually be used to address the table. For each pair of output bits, a set of 6 inputs was selected from 13 possible inputs by one set of configuration registers.

Figure 2.5 shows the PLA for the memory mat. The 48 entries in this lookup table were initially tediously programmed by hand. Bug fixes with this technique or changes to the protocol were extremely cryptic, and often caused new errors. To simplify and enable the correct configuration, a spreadsheet-based programming solution was introduced. Figure 2.6 gives an example screenshot of the programming spreadsheet. The screenshot shows the amount of designer knowledge that was needed for programming the 48 configuration registers for the PLA for a single mat on a tile. This knowledge had to be extended to each of the 16 mats on a tile, as well as to the many other configuration values within the mat. These values were simply documented with the correct configuration for different programming

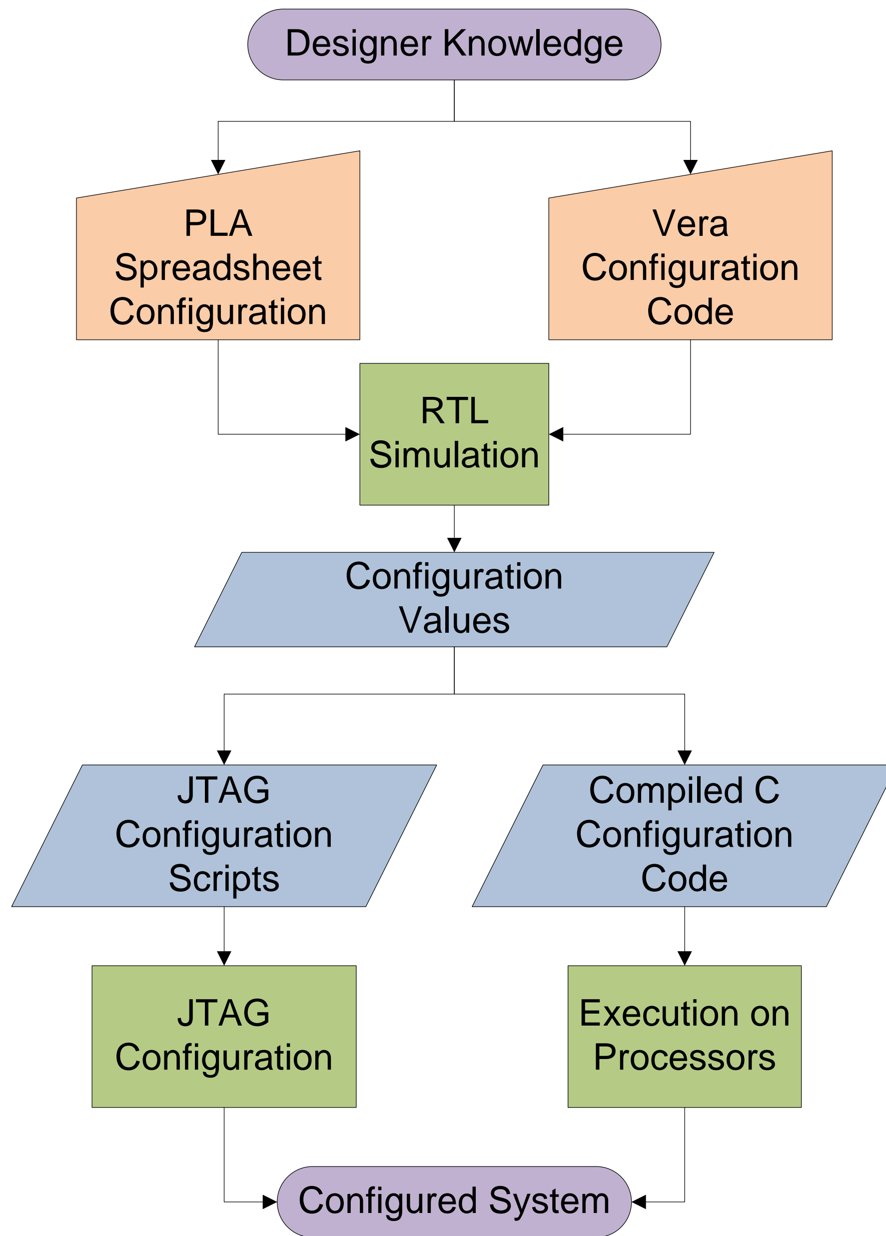


Figure 2.7: The Configuration Flow for Smart Memories was automated and helped efficiently generate cached, streaming, and TCC configurations, once a knowledgeable designer had created generic Vera configuration routines and created programming files for different types of memory mats. A simulation-based approach was used to convert the Vera simulation routines into executable JTAG scripts or compilable C code, which only considered the configuration values and addresses after the configuration had been done in simulation.

```
`endif
    //                                                  MSHR                  1st     2nd
    //                                          MSHR    Unlock  Input   Pipe    Pipe    All     Active
    //-------------------------------------> Free    Index   Sel     Sel     Sel     Pipe    Pipes   LBRead  LBWrite
    ConfigMem[0]                            <= 'b0;
    ConfigMem[`D_PROC_UNCACHED_ACCESS]      <= {1'b0,   1'b0,   1'b0,   2'd3,   2'd3,   1'b0,   2'b01,  1'b0,   1'b0};
    ConfigMem[`D_NET_UNCACHED_ACCESS]       <= {1'b0,   1'b0,   1'b0,   2'd3,   2'd3,   1'b0,   2'b01,  1'b0,   1'b0};
    ConfigMem[`D_DMA_UNCACHED_ACCESS]       <= {1'b0,   1'b0,   1'b0,   2'd3,   2'd3,   1'b0,   2'b01,  1'b0,   1'b0};
    ConfigMem[`D_PROC_ACK]                  <= {1'b0,   1'b0,   1'b0,   2'd2,   2'd0,   1'b0,   2'b01,  1'b0,   1'b0};
    ConfigMem[`D_CRITICAL_WORD_ACCESS_REQ]<= {1'b0,   1'b0,   1'b0,   2'd2,   2'd2,   1'b0,   2'b01,  1'b0,   1'b0}; // When→
    ConfigMem[`D_CRIT_WORD_ACCESS_REQ_CTM]<= {1'b0,   1'b0,   1'b0,   2'd2,   2'd2,   1'b0,   2'b01,  1'b0,   1'b0}; // When→
    ConfigMem[`D_PARTIAL_REFILL]            <= {1'b0,   1'b0,   1'b1,   2'd2,   2'd2,   1'b0,   2'b01,  1'b1,   1'b0};
    ConfigMem[`D_UNCACHED_CANCEL_CC]        <= {1'b0,   1'b0,   1'b0,   2'd3,   2'd3,   1'b0,   2'b01,  1'b0,   1'b0};
    ConfigMem[`D_UNCACHED_INT_CANCEL_MC]    <= {1'b0,   1'b0,   1'b0,   2'd3,   2'd3,   1'b0,   2'b01,  1'b0,   1'b0};
    ConfigMem[`D_UNCACHED_NET_CANCEL_MC]    <= {1'b0,   1'b0,   1'b0,   2'd3,   2'd3,   1'b0,   2'b01,  1'b0,   1'b0};
    ConfigMem[`D_UNCACHED_CANCEL_REPLY]     <= {1'b0,   1'b0,   1'b0,   2'd3,   2'd3,   1'b0,   2'b01,  1'b0,   1'b0};
    ConfigMem[`D_CACHE_REFILL]              <= {1'b0,   1'b0,   1'b1,   2'd2,   2'd2,   1'b0,   2'b10,  1'b1,   1'b0};
    ConfigMem[`D_WRITEBACK]                 <= {1'b0,   1'b0,   1'b0,   2'd2,   2'd0,   1'b0,   2'b01,  1'b0,   1'b1};
    ConfigMem[`D_H_WRITEBACK]               <= {1'b0,   1'b0,   1'b0,   2'd2,   2'd0,   1'b0,   2'b01,  1'b0,   1'b1};
    ConfigMem[`D_I_WRITEBACK]               <= {1'b0,   1'b0,   1'b0,   2'd2,   2'd0,   1'b0,   2'b01,  1'b0,   1'b1};
    ConfigMem[`D_SPEC_LINE_READ]            <= {1'b0,   1'b1,   1'b0,   2'd1,   2'd0,   1'b0,   2'b01,  1'b0,   1'b1};
    ConfigMem[`D_CTOC_XFER]                 <= {1'b0,   1'b1,   1'b0,   2'd1,   2'd2,   1'b0,   2'b11,  1'b1,   1'b1}; // 1st →
    ConfigMem[`D_EWB_CTOC_XFER]             <= {1'b0,   1'b0,   1'b0,   2'd1,   2'd2,   1'b0,   2'b11,  1'b1,   1'b1}; // 1st →
    ConfigMem[`D_EWB_SPEC_LINE_READ]        <= {1'b0,   1'b0,   1'b0,   2'd1,   2'd2,   1'b0,   2'b11,  1'b0,   1'b1}; // 1st →
    ConfigMem[`D_CTOC_XFER_CWB]             <= {1'b0,   1'b1,   1'b0,   2'd1,   2'd2,   1'b0,   2'b11,  1'b1,   1'b1}; // 1st →
    ConfigMem[`D_EWB_CTOC_XFER_CWB]         <= {1'b0,   1'b0,   1'b0,   2'd1,   2'd2,   1'b0,   2'b11,  1'b1,   1'b1}; // 1st →
`ifndef CC_REDUCED
    ConfigMem[`D_LINE_READ_MC_REPLY]        <= {1'b0,   1'b1,   1'b0,   2'd1,   2'd0,   1'b0,   2'b01,  1'b0,   1'b1};
    ConfigMem[`D_MEM_READ_GATHER_REPLY]     <= {1'b0,   1'b0,   1'b0,   2'd3,   2'd3,   1'b0,   2'b01,  1'b0,   1'b1};
    ConfigMem[`D_MEM_WRITE_AND_ACK]         <= {1'b0,   1'b0,   1'b0,   2'd0,   2'd3,   1'b0,   2'b01,  1'b1,   1'b0};
    ConfigMem[`D_INQUAD_GATHER]             <= {1'b0,   1'b0,   1'b0,   2'd0,   2'd3,   1'b0,   2'b11,  1'b1,   1'b1};
    ConfigMem[`D_INQUAD_SCATTER]            <= {1'b0,   1'b0,   1'b0,   2'd0,   2'd3,   1'b0,   2'b11,  1'b1,   1'b1};
    ConfigMem[`D_MEM_WRITE_SCATTER_REPLY] <= {1'b0,   1'b0,   1'b0,   2'd3,   2'd3,   1'b0,   2'b01,  1'b1,   1'b0};
    ConfigMem[`D_MEM_READ_SCATTER]          <= {1'b0,   1'b0,   1'b0,   2'd0,   2'd3,   1'b0,   2'b01,  1'b0,   1'b1};
    ConfigMem[29]                           <= 'b0;
    ConfigMem[30]                           <= 'b0;
--\**  data_dispatch_configmem.v    45% L91    (Verilog)------------------------------------------------
```

Figure 2.8: Screenshot of example code for the Smart Memories protocol controller Code. This is just a small part of just one of the dozen or so configuration (microcode) memories inside the protocol controller.

models, then never changed. In addition to the configuration within a mat, the the memory mats were connected to create caches or higher level structures. In order to enable the verification of the system, several set connectivity patterns (i.e. known cache sizes, sharing patterns, and associativities), were established. The configuration for connecting the mats to enable these caches were then calculated and saved.

Each tile (apart from the mats within) also needed a great deal of configuration in order to implement anything other than a default uncached memory configuration. The configuration process was achieved either by code executed by the processors (on startup, one processor could run code in an uncached mode which would configure the chip into a cached mode), or it could be done before boot via JTAG writes. Therefore, the challenge of configuring the tile reduced to determining what configuration values needed to be written to what configuration addresses, and in what order. This was accomplished using a simulation-based flow to generate JTAG and C scripts from higher-level Vera configuration routines, illustrated in Figure 2.7.

The mechanism for actually configuring the protocol controller was much the same as for the tile. However, the configuration "registers" in the protocol controller were more

substantial than those in the tile. Rather than a few scattered registers, the protocol controller's programmability was provided via large programming memories. These looked like giant microcode banks. The code (simulation Vera) that configured them looked very similar to the tables as implemented in hardware, and was so large and verbose that it often crashed text editors trying to modify a single bit to fix a bug. A small example of this massive body of code is shown in Figure 2.8. The code that configured them was hand-tweaked, and required the knowledge of the designer in order to know when a bit should be set to 1 or 0. Most of the configuration was either single bit selectors ("do I source this field from the previous block, or from a state register?") or type selectors ("send out a message of type `X`"). In many cases, the configuration memories were even more complex, because they were implemented as two linked memories in the form of a TCAM (Tertiary Content Addressable Memory). The first compared against some masked lookup data to give a matching index, and the second table provided the output values at that index.

The mechanism for configuring the off-chip memory controller was very similar to that of the protocol controller's microcoded, TCAM-based lookup system. In the same way as the protocol controller, while the configuration values mapped to addresses in configuration space that were written by the software flow shown in Figure 2.7, the actual determination of the values in the Verilog script was written by hand.

2.3 Programming Limitations of Smart Memories

Smart Memories was fabricated (Figure 2.9), and was brought up into a fully functional system [44]. The programming flows were adapted from the verification stage to generate configuration tests in JTAG. It used the same flow intended for generating C configurations to generate the C code the hardware ran to configure itself. While no functional errors were found in the actual hardware, a plethora of errors arose due to the configuration scripts and the tools that generated the scripts. The scripts were also brittle in that they had never been tested for more than a processor or two, notably not for theoretically possible but untested scenarios, like the configuration of multiple quads from a single running quad into a TCC configuration. While these were ultimately debugged, it was a hand-done process and took time and thought.

Since the Smart Memories system was a fully functional programmable memory system,

Figure 2.9: Smart Memories Die Photo. The chip was packaged and put into a 4-quad system with 32 processors.

we presented it to researchers in other groups/at other institutions, as a configurable memory system, asking "What can Smart Memories do for you?" There was enthusiasm and curiosity from other researchers, in particular, a group from Princeton that had been proposing modified cache coherence protocols in order to prevent cache timing attacks [46,47,45,10,9]. They were having trouble convincing the circuits/architecture/hardware community that their approach was viable. Their simulations showed that their solution did a good job of foiling cache timing attacks, but those in the hardware-related fields did not believe that it was possible to build such a design in an efficient way. These secure-cache designers saw in Smart Memories a potential tool for doing real hardware simulations, by configuring it to implement their protocols, rather than building a CMP from scratch.

The simplest security protocol [47], required just the addition of a "Lock" bit to the cache lines to prevent certain cache lines from being evicted (if all ways of a set were locked, then none could be evicted and the cache line would have to be just brought in uncached from off chip every access). This sounded simple and was exactly the sort of thing Smart Memories should have been been able to do.

The first consideration was the lock ('L') bit and where it would fit into Smart Memories' hardware. To control the cache behavior based on the L bit and to consider it in the eviction protocol, it had to be included in the programmable 'meta data' bits of the tag mat. This issue was addressed by filling out the memory mat PLA calculator accordingly, which was simple enough to do. However, system and protocol controller configuration tasks were drastically more complicated. We had to write new tasks for configuring the cache in 'PL-Mode', code for each tag mat, data mat, etc. The code for these tasks were each similar to the obscure code shown in Figure 2.8, which had to be hand-modified to do what we wanted. The Protocol Controller (PC) code handled the control of eviction, which had to be modified to make sure the eviction candidate was not in the 'L' state, and updating the 'L' state accordingly given special operations from the processor. This hand modification was nearly impossible, and required several sessions with the designer and author of the protocol controller in order to configure it. Even with the designer present it was very unclear what each bit should be (the generally successful strategy was to find a command that was "pretty close" and copy that line of the configuration). Finally this was done successfully for the simplest design, PLCache [47]. The more advanced cache designs that the security researchers wanted to implement, could not be realized on the taped-out Smart Memories hardware.

As we were developing Smart Memories, we realized that there was no reason that the chip, while fully configurable in design, had to be taped out in a fully configurable format. Configuring a flexible chip after fabrication was very difficult, and we could focus our energy on ways to make sure we configured the chip correctly for a given application, and manufacture only one specific configuration at a time. What we would ideally want to do is create a high level specification of the protocol and hardware we want to implement, then from that configuration generate hardware. We knew that generating hardware from scratch from a specification is an impossible task, so we focused on building a flexible hardware template that can take the specification in order to configure itself.

This realization led us to consider a new methodology for designing chips, which we refer to as a *generator* methodology [39]. The inherent flexibility of this method lends it to a good solution for our problem of implementing and experimenting with new memory protocols. However, the problem that we had in Smart Memories of how to easily and reliably configure such a system remains, so we must ensure that the generator for this application is able to take a higher level specification as configuration input.

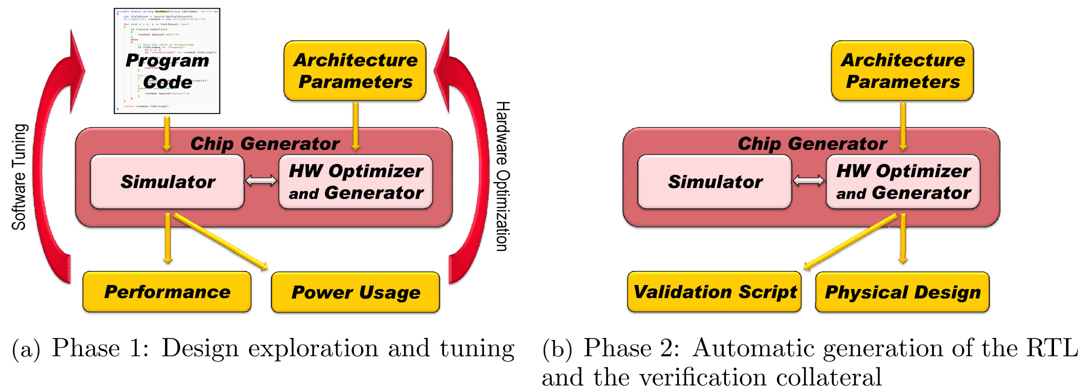
2.4 The Generator Approach

In order to make it easy (or at least possible) to specify a new memory protocol, we want to start with a higher level specification of a protocol rather than having to describe the value of every bit in a look up table which implements a protocol. In order to create such a specification, however, there needs to be an abstract hardware execution model associated with it – we need to know what our memory system primitives are that we can compose and modify to implement the protocol. The higher level specification would "program" this hardware, to create an implementation of the protocol. This does require creating a template of the hardware needed in a memory system that can accept the configuration program. The generator methodology described in this section provides a mechanism for creating a such a programmable hardware template.

The generator methodology creates a system that embeds knowledge about design options, dependencies, and optimization tools inside the design. Rather than record one output of the design and optimization process—the design produced—the *process* is codified, so future "designers" can leverage it for other designs with different system constraints. The design artifact produced becomes *the process of creating an instance*, not the instance itself.

The design becomes a *generator* system that can generate many different platforms, where each is a different instance of the system architecture, customized for a different application, a different design constraint, or a different memory protocol.

A generator provides application designers with a new interface—a system level simulator whose components can be configured and calibrated. In addition, it provides a mature software tool chain that already contains compilation tools and runtime libraries, since even though the internal components are configurable, the system architecture is fixed and some basic set of features always exists.



(a) Phase 1: Design exploration and tuning (b) Phase 2: Automatic generation of the RTL and the verification collateral

Figure 2.10: Two-phase hardware design and customization using a chip generator: (a) Tight feedback loop of both application performance and power consumption enables fast and accurate tuning of design knobs and algorithm. (b) Automatic generation of hardware to match the desired configuration.

Configuring the system becomes a two phase process (Figure 2.10): In the first phase, the designer tunes both the application code and the hardware configuration. The generator's system simulator provides crucial feedback regarding correctness and performance, as well as physical properties such as power and area. The designer can therefore iterate and quickly explore different architectures until the desired performance and power or area envelope is achieved. Once the designer is satisfied with the performance and power, the second phase further optimizes the design at the logic and/or circuit levels and generates hardware based on the chosen configuration. Furthermore, it generates the relevant verification collateral needed to test the chip functionality (since all tools can have bugs).

2.4.1 Building a Generator for a Memory System

Given the generator methodology and our problem of building a programmable memory protocol system, we need to define the “template” that we will program to implement different protocols. This template needs to address the first problem that we faced in our earlier flexible system: the logical submodules need to be able to easily access different logical inputs and signals, for control logic decisions and for manipulating and communicating data. We also want to leverage the power of the generator to allow different numbers, sizes, latencies, etc, of units. To that end, Chapter 3 goes into detail on the type of template one might construct for an overall hierarchical memory system. Since the generator idea encourages the idea of hierarchical templates, we also go in depth in one of the controllers of the system and explain how it is parameterized, and again another level deeper for the individual blocks which make up the controller and show how those too are generators.

The inflexibility of the hardware was not the only issue we faced in the previous architecture. It was too difficult for a human user to program by hand, and the tools that did exist for programming it were rudimentary and ad-hoc. Therefore, we also need a language for specifying the protocol which we would like to implement with the flexible hardware architecture. Chapter 4 describes the high level programming methodology that helps to configure the hardware template and makes it possible to implement standard and novel memory protocols with the same template. Before we can address the problem of how to specify a protocol in such a way that makes it possible to implement and test in hardware, we should consider the different levels at which one could specify a protocol, which is the topic of the next section.

2.4.2 Protocol Specification Levels

Figure 2.11 shows the different levels at which we can consider specifying protocols. The *Protocol Specification Level* is the most abstract level, and is completely implementation agnostic. Rather than saying anything about how the protocol should be implemented, it states basic properties about its observed behavior. For memory protocols, certain sets of such rules are common and are grouped together by standard, agreed-upon names. For example, the protocol could be specified as implementing Total Store Ordering [38] or Sequential Consistency [25], or could make some security guarantees about interaction between

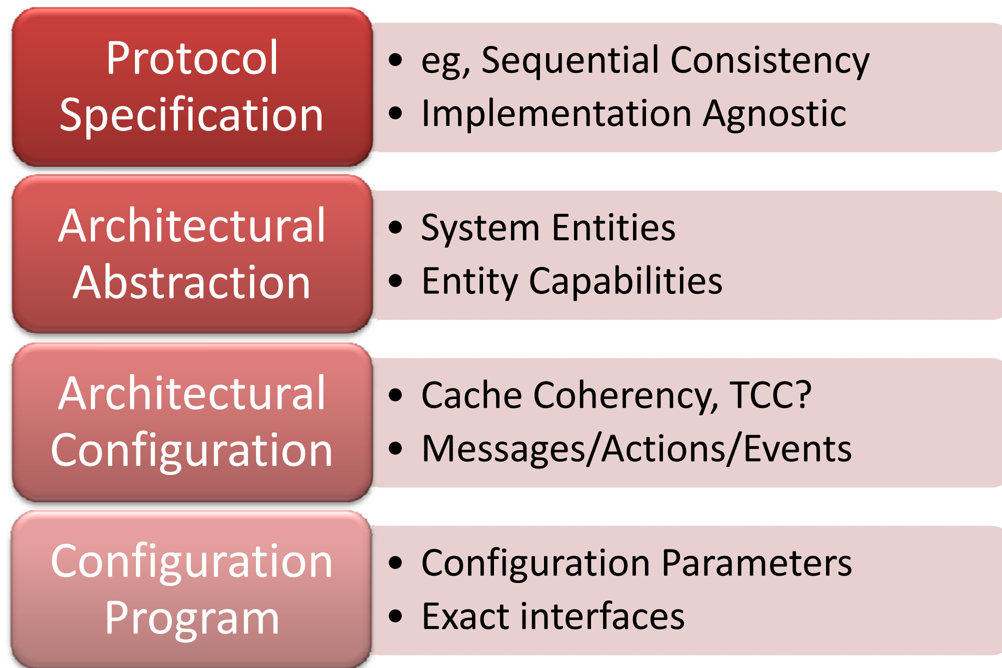


Figure 2.11: Memory System Specification Levels

processes. The protocol level is the level at which we should perform a good deal of verification, because adherence to this protocol is what we actually want to test. While writing rules and building a checker at this level can be tricky, Chapter 5 describes a successful technique for doing so.

The protocol level does not provide any information about how to actually implement the desired protocol. Moving closer to the implementation, the *Architectural Abstraction Level* begins to make assumptions about the underlying architecture. For example, it might refer to what entities exist (e.g. processors, caches, communication channels), and describe things that these entities can do (for example, main memory can be read and written given an address, caches can modify their contents based on their current state and a given operation, processors can stall, etc). Current generator tools work at this level, and parameters at this level are used to configure the system. A good understanding of this architecture, its capabilities, and its configuration requirements, is essential for using the architecture to implement a memory protocol, but does not yet provide details about a specific protocol.

The *Architectural Configuration Level* describes how to configure the agreed-upon architecture to implement the desired protocol. There are many well-studied memory protocols which can implement sequential consistency, such as MESI [32] or Dragon [5]. This level describes the desired (lower-level) protocol in some language that is understandable and implementable by the architectural template above.

The *Configuration Program Level* uses the program from the previous level to actually generate the configuration information for the virtualized hardware in the chip generator. This could mean populating microcode memories, selecting parameters to control the template, or selecting from appropriate subinstances, depending on the type of generator used.

Many specifications have been proposed to verify languages at the protocol level. One higher level description is a protocol's *message flows* [43]. A message flow is the sequences of messages sent among processors to complete a single transaction in the protocol. Talupur et. al. use the flow description of a protocol to construct invariants for verification, but researchers could also leverage the same description to generate hardware programs or configurations. However, this is too high level to easily translate into hardware, especially without a clear architectural level model. For implementation purposes, we choose to specify protocols at the architectural configuration level, and use the hardware template to provide the necessary information at the architectural abstraction level.

The University of Wisconsin's Specification Language for Implementing Cache Coherence (SLICC) [41,29], is a good model of a language at the architectural configuration level. It was a language used by the GEMS Multifacet Simulator for its shared memory protocols. It provided a way to more intuitively and compactly describe cache coherence protocols, by describing them in a C-like language which provided the following constructs:

- State machines
- States (enumerated per state machine)
- Events (enumerated)
- Actions (enumerated and described)
- Transitions (described)

In addition, SLICC has strong primitives for message types and queues between different state machines, as well as providing support for external types and methods (which are

written outside of the SLICC language). SLICC was designed for generating simulator code for modeling hardware performance and correctness, and also benefited the verification of a protocol by providing easy to analyze formats as output.

Given that we'd like to start with a protocol description similar to the one provided by SLICC, but compilable into hardware, we need to build hardware that can accept such a specification as input. The next chapter describes the method we have used to create the hardware platform to take an architectural configuration specification. The specification language itself is discussed in detail in Chapter 4.

Chapter 3

A Programmable Hardware Architecture

We prefer to specify memory protocol implementations at a high level, to avoid becoming lost in the implementation details. Therefore, we need a form of hardware that can take as input the program for the memory protocol, and have the program take into account the high-level memory architecture. This chapter starts by describing the architecture of a modern hierarchical memory system. This background makes it clear that our protocol "program" consists of many concurrent threads working in parallel. Section 3.2 describes the types of functions that these threads of control need to perform, then Section 3.3 gives a quick overview of the architecture that these threads run on. Section 3.4 then describes the hardware for the most complex thread, the protocol control, in more detail. This example demonstrates how we make that controller programmable by composing flexible primitives in a structured template.

3.1 A Hierarchical Memory System

A modern memory system is hierarchical. While there are many possible configurations, and they could be flexible in a variety of ways, by setting a basic architecture template, the protocol can be specified for many implementations of that architecture. Our basic architecture consists of both on-chip and off-chip components. The on-chip components consist of one or more units which originate commands (traditionally processors), some local memory, and the communication between them. The on-chip components also include

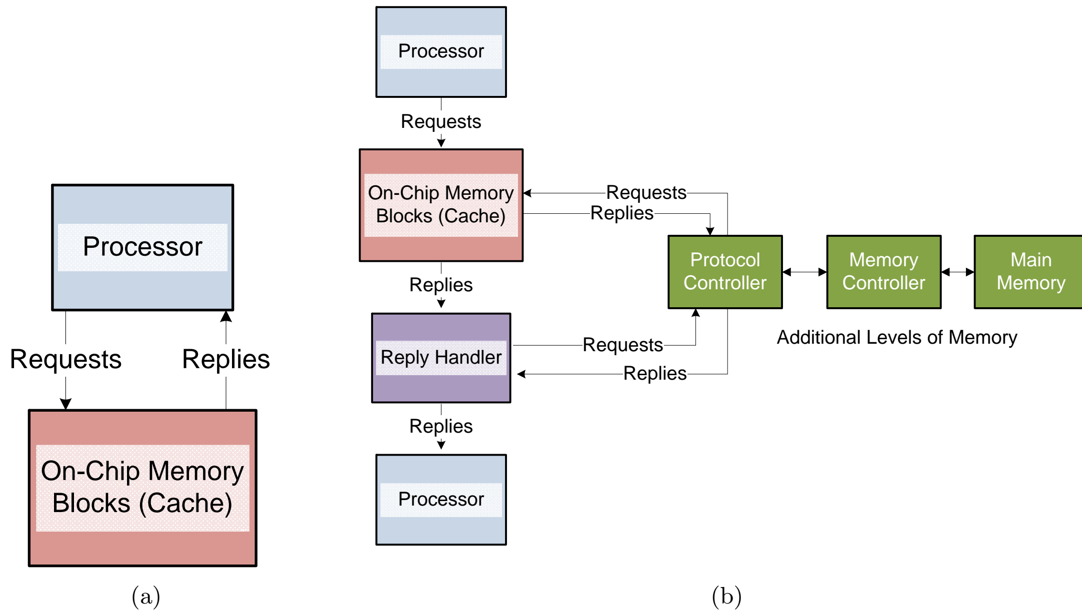


Figure 3.1: A simple memory system architecture. a) A single processor communicating with an on-chip memory block by sending it requests and receiving data in reply. b) The same system, but unrolled to show more detail. The processor on the bottom is the same processor as the one at the top of the figure, but a reply handler actually interprets return codes of the memory block, and is able to communicate to additional levels of the hierarchy if necessary to satisfy requests. It is able to communicate with an on-chip protocol controller, which in turn is able to communicate with both the off-chip components and the on-chip memory blocks to take the steps necessary to satisfy the processor's request.

a module for interfacing with off-chip components, which can include interfaces to other chips, the network, or larger, slower memories. Section 3.3 describes this architectural template in more detail.

Figure 3.1 shows an abstraction of a simple hierarchical memory system. Figure 3.1.a shows the most basic concept, where a processor (CPU) sends requests to a local memory and receives data or other replies back. Figure 3.1.b shows a somewhat more realistic "unrolled" view of the complex system, where a CPU sends requests to the memory system, which arrive first to a local memory block (often, a cache or scratchpad). A reply handler analyzes the result from the memory block, either returning the data to the CPU to allow it to continue to run, or stalling the processor before it tries to use the value returned from the load, while the reply handler takes the steps necessary to satisfy the request.

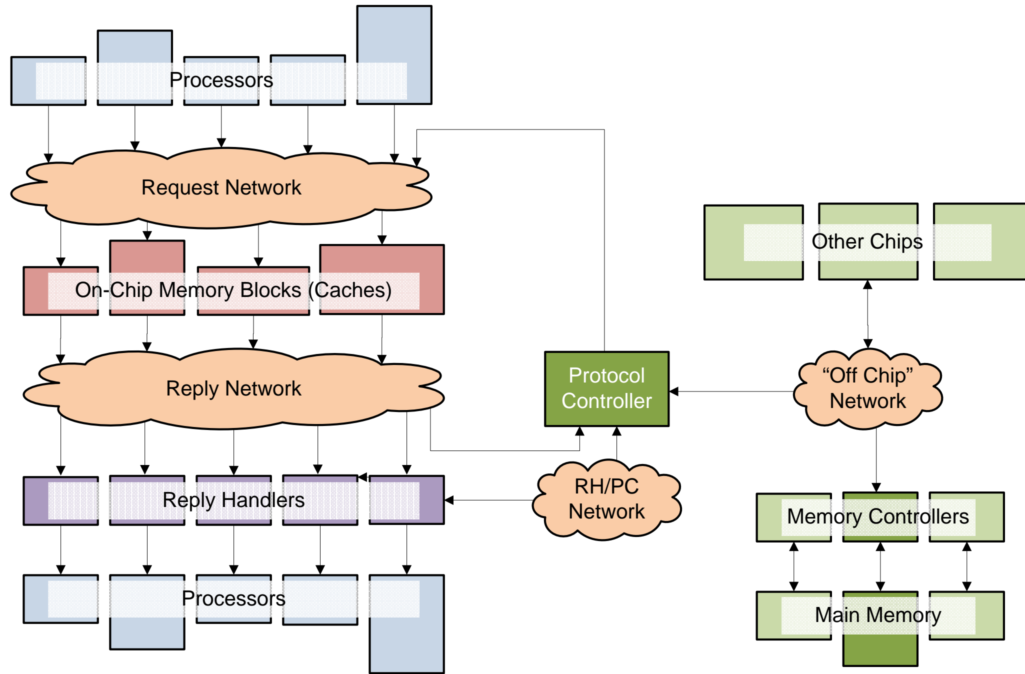


Figure 3.2: A chip multiprocessor memory system architecture, with heterogeneous processors and memory blocks.

Because not all of the data needed by an application or system can be stored on-chip, modern memory systems must also interface with off-chip memory components. If a processor request can not be satisfied locally, then the reply handler communicates with a more powerful protocol controller. The protocol controller can query and update the memory block, as well as communicate off chip to a memory controller which has access to off-chip and/or networked memory.

This simple view of the hierarchical system becomes more complicated, in a system with multiple processors, as shown in Figure 3.2. Multiple processors are capable of sending different types of messages into the memory system. A network is now required to arbitrate between the different processors to send their requests to the correct destination(s). The memory blocks can be configured as caches of different sizes and capabilities, or entirely different structures. Each reply handler is associated with a processor, so a reply network reflects the request network to route the responses back to the originating processor. The reply handlers again communicate to the protocol controller, which now must arbitrate between the different requests from the reply handlers, enforce ordering between them, and

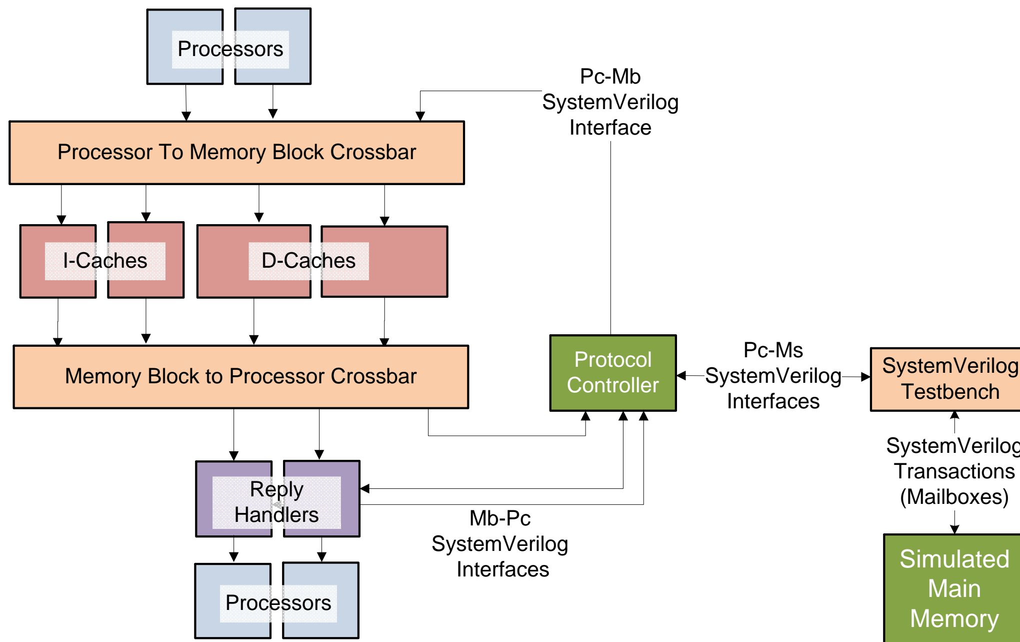


Figure 3.3: One instance of a CMP template, including the components for the memory system. A parameterized number of processors connect to memory blocks through a crossbar. The crossbar's width is parameterized, but its structure is known. The memory blocks implement caches and their behavior is specified in the higher level language. They communicate back to the processors through reply handlers, which on some operations communicate to the Protocol controller through dedicated direct interfaces (each reply handler has its own connection to the Protocol Controller). The protocol controller communicates to the memory blocks in the same way as a processor, through the crossbars. In this instantiation, the protocol controller has dedicated interface to the off-chip memory system, which is implemented in SystemVerilog simulation testbench.

send requests into the request network similar to the way the processors do. The protocol controller still sends requests off chip but these may go into a more sophisticated network with multiple memory controllers, and even other chips' protocol controllers for a fully hierarchical system.

To minimize complexity and increase clarity, we will use this architecture for the rest of this thesis. For performance, many systems add additional levels of memory hierarchy. For example, processors today have 2-3 levels of on-chip caching, with the level one caches generally private and the 2nd or 3rd level shared. The hardware still fits into the template in Figure 3.2, though we would need multiple, more complex controllers to manage the interactions between the memories. For this purpose we would still describe these as memory blocks, and introduce more controllers to send them requests and interpret their replies. The result would look similar to the memory blocks and associated protocol controller shown in Figure 3.2, but rather than communicating directly with the processing elements, these controllers would communicate between the first level protocol controller and off-chip, and with their own associated memory blocks which maintain the additional cache levels.

To understand what all these blocks are doing, the next section reviews how the whole system works together to provide a simple view of the memory system from the processors' perspective. This description is a known as a *memory protocol*.

3.2 Describing a Memory Protocol

One of the main advantages of a hardware system is that many things can be happening in parallel. In order to maximize this concurrency and increase the locality of the system (to make routing and wiring shorter and faster at all levels), we want to have several blocks working relatively independently, rather than having one holistic controller which orchestrates every interaction in the system. As we described in the previous section, those blocks are macro-level entities like processors, memory blocks (caches), protocol controllers, the off-chip network, and main memory. In the same way, these macro level entities may be divided into micro level blocks, e.g. the different stages inside each of the larger controllers. The protocol controller might have different subsystems to handle communication with main memory vs. communication with memory blocks.

In our model, each block (or subblock) communicates with a subset of the other blocks. This communication is done via generalized messages. A message could be as simple as

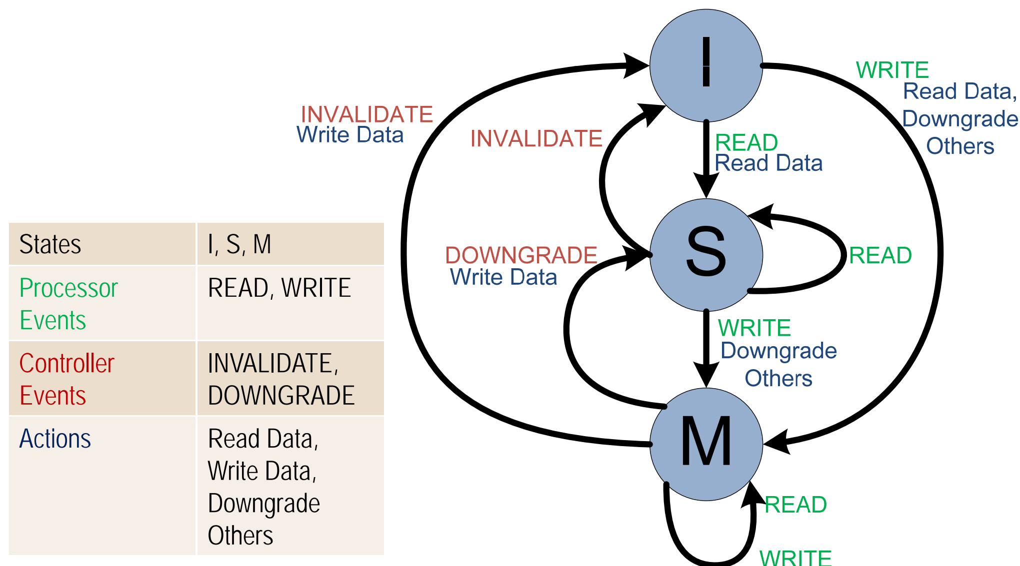


Figure 3.4: A simple MSI memory protocol, demonstrating the basic components of a shared memory protocol.

an instruction of what the neighboring block should do, or an acknowledgement that the instruction has been completed, but the idea can be extended to include any number of additional data or flag fields (Address, Data, State, and so on).

At the leaves of this hierarchical design, the internal behavior of each of these blocks is not complicated, and is easily described with a finite state machine (FSM). Each block tracks some number of operations and maintains the state and associated data for each of these operations. In addition, it selects the correct operations to perform at a given time based on an incoming message from another block, and the current state of the associated operation. Local memories usually have state associated with each data entry to maintain the state of a given piece of data. In order to determine the correct action(s) to take when a message is received, the hardware can query the state. Once the current state is determined, the hardware can update the state and perform zero or more actions.

The mechanism for the data's state modification requires a number of fine grained updates resulting determined by the received message and the actions performed. These actions include receiving data, moving data around, translating the data, and composing messages to send out to other blocks. A key capability of the hardware is the movement of pieces of data ("blocks" or "lines"). The hardware provides mechanisms to move pieces

of data around, such as allowing it to be copied from an interface or input buffer into an internal storage structure, and again from the internal storage structure to output ports or buffers. This action can be described at a large granularity, for example, by describing the actions on an entire block of data, but the actual hardware may need to be able to serialize this action in order to make it implementable.

To support and track this data motion, each entity maintains its own internal state. Internal storage buffers data that is passing through and tracks outstanding operations, because requests can not be satisfied in a single operation. Common structures of this type include the Miss Status Handling Register (MSHR), which tracks a more complex state than the simple line state (for example, which processors are waiting for the miss to be satisfied), and the Line Buffer, which stores large blocks of data as they are pass through the system and accumulate.

So far we have focused on control and sequencing operations, since that is the majority of what these blocks do. Of course, some blocks also must manipulate the data fields, e.g. to create the hashed cache index in ZCache or RPCache, or to compute the eviction candidate. Our architectural template allows the protocol to specify arbitrary logical functions. In these cases, the hardware generator provides a clean interface to attach functional units which are controlled by the FSM.

The procedure for maintaining and updating state, and describing the actions performed in a given state when certain events occur, is formally described with a memory protocol. Figure 3.4 shows a toy memory protocol, which identifies the basic building blocks of a shared memory protocol. First, cached addresses in the system ("cache lines") have associated *state*. In this simple example, these states are Invalid, Shared, or Modified. The state of an address gets modified based on certain *events*, such as messages coming from a processor or from another chip via the network. These *transitions* depend on the current state of the address and the observed event. Finally, while making the transition, the system needs to perform certain *actions* in order to actually correctly make the transition.

What is hard about designing and implementing such a protocol is making sure that all the pieces work together. Adding the special address hash for a new cache design is not complicated. However, describing all the message actions for transactions, or cache consistency, is where the errors are made. Therefore, we want to create primitives that cover the communication and sequencing part of the controller hardware, and allow users to add special hardware functional units for the tasks that are easier to describe. The next

section will walk through the hardware constructed to build a generalized controller that can not only implement this simple protocol, but is also flexible and programmable enough to implement novel, more complicated, protocols.

3.3 High-Level System Template

To fully specify a hardware memory system, therefore, we need an overall abstract architecture. We would first need to specify the number of levels in the hierarchy, and the sharing patterns at each level. For shared caches we need to define the communication mechanisms between the different controllers which access them, so that we can ensure the shared caches remain in a consistent, coherent state. The protocol can assume an abstract way of passing messages, but the actual mechanisms for interconnect (e.g. a direct, fully parallel interface, a flit-based network protocol, or an arbitrated crossbar) need to be specified inside the hardware template. Since communication networks are a well-studied problem, we can use existing generators for those. However, each additional level of hierarchy requires new controllers to implement their part of the protocol. Rather than building a new controller from scratch for every level of the hierarchy (though they may have different possible actions and tracking structures), we will create an abstract template for a controller and configure it for the job it needs to do via the configuration program.

Therefore, the next section will focus on how to build a template for a single abstract controller in the system, where the system is made up of multiple controllers connected by interconnections of various types. We will discuss in detail the micro-templates which are used as building blocks inside the protocol controller for a shared cache, and how the micro-template is made programmable. For this purpose we will focus on the protocol controller which interfaces between main memory and a number of shared caches, the same as the protocol controller depicted in Figure 3.2 and Figure 3.3.

3.4 Generalized Protocol Controller

We'd like to build a memory system which can implement a protocol such as the one shown in Figure 3.4. As we mentioned earlier, the system will be made up of a hierarchical system of interacting, independent blocks. We are able to generate both the internal controller hardware and the needed configuration of this hardware by leveraging a set of configurable

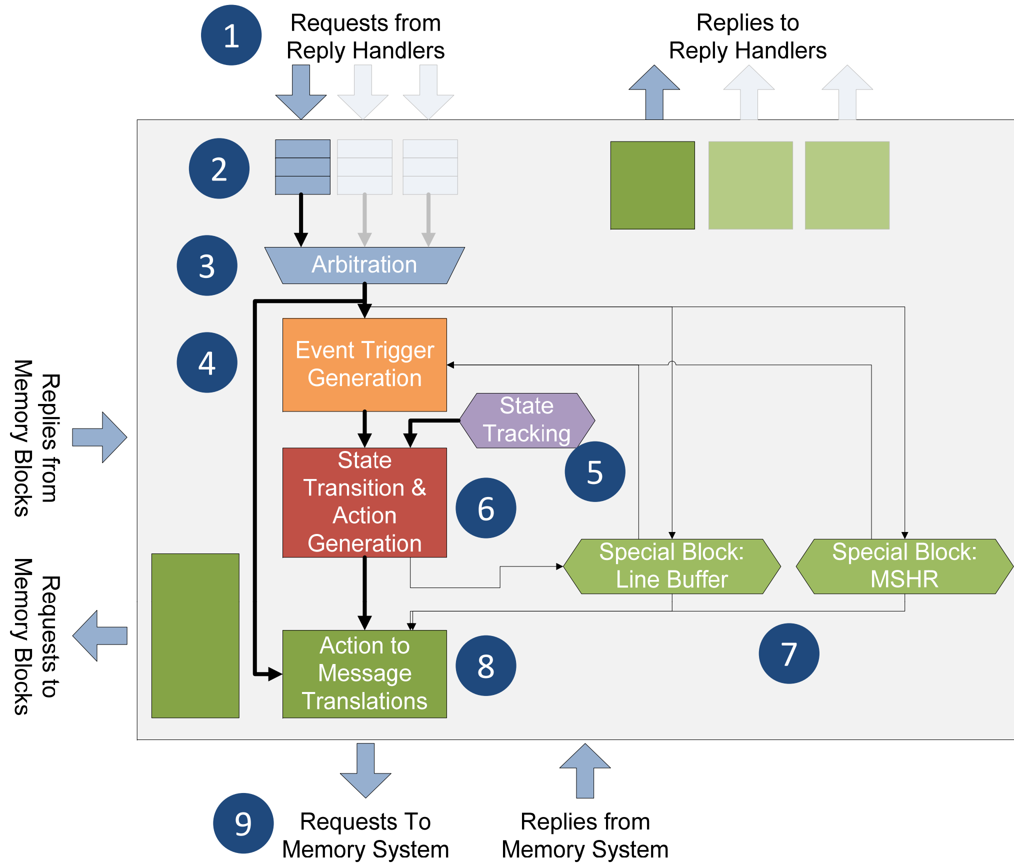


Figure 3.5: A Generalized Protocol Controller, showing the different flexible components which can implement an arbitrary protocol. These subcomponents are discussed in detail in this section. ① depicts an input interface on which a message is received from another block in the system. ② is a set of FIFO structures used to buffer the incoming messages. ③ is an arbiter to select between incoming messages. ④ is a subunit which generates events based on the incoming messages. ⑤ is a tracking unit to hold the current state of transactions in the controller. ⑥ converts the triggered event and current state into a set of actions. ⑦ depicts special blocks, in this case a Line Buffer and an MSHR, which can be controlled or queried by the actions. ⑧ translates the actions and other input data into outgoing messages. ⑨ shows the output buffers to blocks in other units.

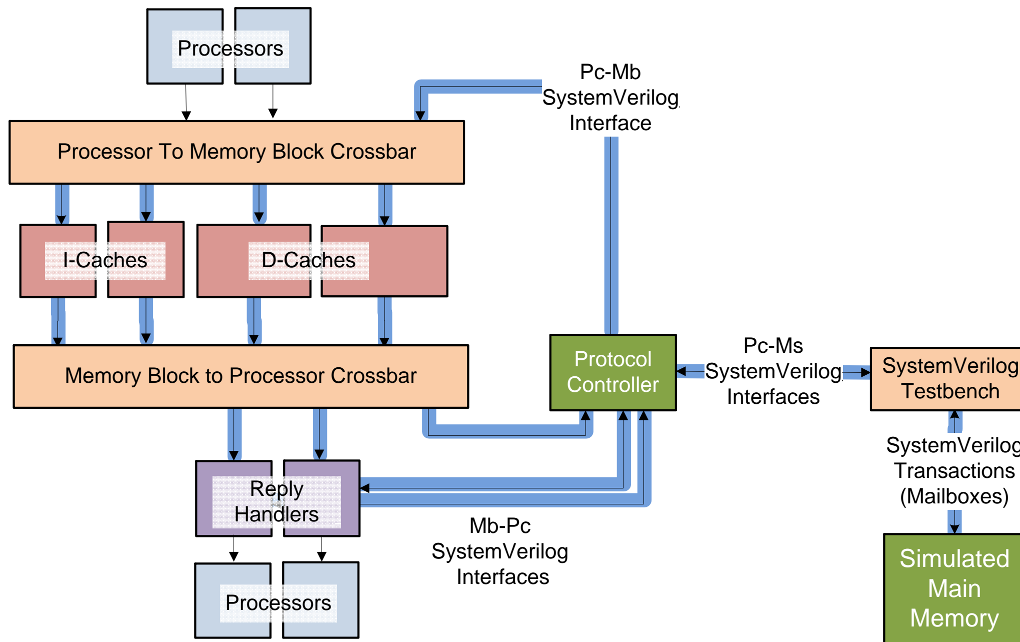


Figure 3.6: The hardware interface primitive provides an encapsulation of a set of signals and widths, and provides a clean way to specify the interfaces between units. It is a parameterized version of a SystemVerilog interface, where the parameters describe the signals and their widths. The crossbar can reference the same interface object to query about its signal names and widths, in order to create the crossbar internal structure and interfaces.

building blocks. These building blocks lie at the core of most controllers, and are described in the remaining part of this chapter. To help better understand where and why these blocks are used, we will describe them in the context of creating a protocol controller that interfaces caches to the next level in the memory system. To create the template for this controller, we used Genesis [36, 35], which allows powerful, hierarchical parameterization of Verilog modules and allows the template to be customized for many applications. Throughout this section we explain how we parameterized each low-level hardware piece to provide this flexibility. We start with the connections between each of the individual controllers.

3.4.1 Interfaces

In order to implement any protocol in a system like the one in Figure 3.2, the protocol controller needs to interact with the reply handler(s), memory block(s), and the off-chip memory system. The types of messages sent to and received from each neighboring block

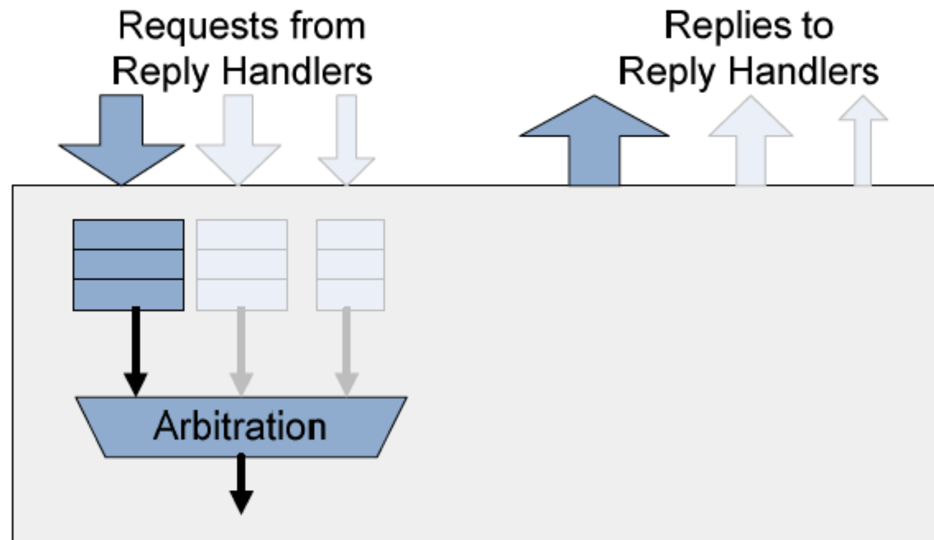


Figure 3.7: Additional uses of the hardware interface primitive, zooming in on ①, ②, and ③ of Figure 3.5. The same primitive is used to parameterize the FIFOs used to buffer the incoming messages, as well as to build an arbitration mux specially sized for the incoming data.

are different, and may also vary depending on the protocol, hierarchy, or data sizes/widths. However, the basic idea of sending messages with a Type and associated data fields remains the same, therefore we design a programmable interface primitive as one of the key parts of a memory system generator and use it in many modules. Figure 3.6 shows the primary use of the interface primitive. The interface primitive compiles to a SystemVerilog interface (which encapsulates the connectivity between two or more modules), and allows Genesis to control the name and width of each signal. Each interface describes the set of signals going in one direction, and a separate interface can be used to describe signals flowing in the other direction. This is the way to build the "message" specification primitive, but it is not the only use of this hardware primitive. Aside from making it cleaner to instantiate inputs and outputs (an inherent benefit of the SystemVerilog interface that is under the surface), the interface provides a packaged way for modules to agree on how they will communicate. Both sides can query the interface object's parameter space to learn about the signals and widths that they must provide or accept. In addition, since FIFOs don't change the interface (they just add delay element), the template can use the same interface specification but decide to insert a FIFO between the communicating units. This FIFO depth decision is orthogonal to

the protocol specification so could be made for other optimization or performance reasons.

The parameters for the interface are :

- $MSG_NAME = name$
- $SIGNALS = [\{name => ..., width => ...\}, ...]$
- $LATE_BIND_SIGNALS = [\{name => ..., width => \text{`}LATE_BIND_...'\}, ...]$
- $LATE_BIND_MODULE = module_name$
- $LATE_BIND_PREFIX = prefix$

The MSG_NAME parameter allows an intuitive naming of the interface, such as "Pc2MbReq". The SIGNALS parameter is a hash of signal names and widths, and is optionally replaced by the LATE_BIND_SIGNALS parameter. If the LATE_BIND_SIGNALS parameter is given instead, then some widths inside the hash may be specified as LATE_BIND_* (for example, LATE_BIND_DATA_WIDTH). This indicates that the width of this signal should come from another source, by default the parent module. If the parent module is not the desired source of the width, then it can be overridden for this interface instance with the LATE_BIND_MODULE parameter, and the LATE_BIND_PREFIX can be used to differentiate signals with the same base name. For example, the parent module may contain the calculation for the P2MB_DATA_WIDTH coming from the processors into the memory block, and also the calculation for MB2XBAR_DATA_WIDTH for the replies from the memory blocks. LATE_BIND_PREFIX can used to distinguish which of these parameters in the parent module should be used for LATE_BIND_DATA_WIDTH.

Because the protocol controller is interacting with a number of different blocks, it will need to buffer incoming messages and arbitrate between them. The interface primitive can be re-used to create familiar hardware blocks, as shown in Figure 3.7. One is a FIFO queue primitive, which can be used to buffer incoming messages into FIFOs of the correct widths[1].

In order to arbitrate between the buffered inputs, the FIFO primitive can be coupled with a parameterized multiplexer primitive to select among a set of input interfaces, and to

[1] [24] has shown that there is great benefit to keeping these sorts of structures narrow, as it allows more efficient state inference propagation across clock boundaries. The information contained in the interface structure allows a set of FIFOs to be constructed in parallel, one for each field in the interface, rather than blindly concatenating signals to be buffered as a single input to a FIFO. This allows more efficient synthesis of the design.

construct the multiplexer output interface as the union of the input signals. The multiplexer also outputs an `Id` signal on the output interface (if it doesn't already exist) to indicate which input it came from.

When the protocol controller wants to compose an output message, it may compose it generically but then only want to send it to a single destination. For example, it may compose a generic reply but then send it only to the reply handler which originally sent it a request. For this purpose, the interface primitive object can be passed around and referenced when creating other hardware blocks. For example, it can compile down into a SystemVerilog interface used to connect with a demultiplexer (demux), which sends the input out onto a given output. Because of the more powerful parameterization of the output interface objects, the demux template can query each of the interface objects to create the parameters for the input interface (as the union of the set of all output signals). Then it can instantiate the correct SystemVerilog (hardware) interfaces on each of its ports, and also save the object reference so that the other blocks which want to connect to it can learn about the signals for the input interface. Internally, a simple demux is instantiated for each signal which makes up the interface, even though the control signals are the same, because it is more efficient for synthesis to have narrow, understandable data paths.

Finally, for times when an input message may need to be delayed by a given number of cycles, (for example, because looking up the current state of the line referred to in the input message takes a fixed but non-zero number of cycles), the interface can be used in a very simple fashion by a delay primitive template. The delay template takes as a parameter a reference to an interface object, then will use that description to create SystemVerilog interfaces at its input and output, then internally create the series of flops needed for each field described in the interface object.

Having described the interfaces at the edge of the controller, we now describe how to implement the flow of data and the control mechanisms within.

3.4.2 Events and Transitions

When the protocol controller receives a message, it should map that message to an event (similar to the types shown in Figure 3.4) so that it can associate the event with its current state and make the correct transition and actions. Therefore, we need to construct some logic for determining the event from the incoming message. The high level goal is to map a set of inputs onto a set of outputs. The logic will do this by comparing some or all of

the fields from the incoming message, and referring to status results from inside the system (such as whether internal state is full or has space available to allocate). When the mapping is very dense (all of the input values are used and each most combinations have a meaningful output), a standard way of doing a translation of this type would be a Lookup Table (LUT). When the mapping is more sparse (not all of the inputs are needed for each output case, and not all the input values have unique outputs), a tertiary content addressable memory (TCAM) is a more efficient structure.

A LUT is a widely used technique for building a simple logical function that translates a single input into a single output. Values that aren't specified in the mapping are "don't cares" or X. These structures can be used for simple things like translating opcodes, but they are used internally to build more complex structures. The parameters for the LUT are:

- $LUT_PARAMS = [\{lookup_val => ..., table_val => ...\}, ...]$
- $LOOKUP_VAL_SIZE = width$
- $TABLE_VAL_SIZE = width$

The LUT_PARAMS parameter gives the mapping from `lookup_val` to `table_val`. Note that there does not need to be a `table_val` for every possible value in the table. The LOOKUP_VAL_SIZE parameter indicates the width of the input of the function, and the TABLE_VAL_SIZE parameter gives the width of the output.

Unfortunately, for most cases, the LUT is not well-suited for mapping incoming messages to events. This is because multiple fields can influence the type of event (for example, one might want to consider the message type, the state of the target line, and the state of the eviction candidate), and the consideration of those fields is not necessarily uniform or complete. In many cases a "don't care" value is required for some of the inputs, and the outputs are not all fully specified either and should take default values. For this reason, we introduce a programmable ternary content addressable memory (TCAM) primitive. The TCAM is an extension of the LUT which maps multiple inputs onto multiple outputs, but allows for very sparse mappings instead of the dense LUT mapping. It allows don't care values on the inputs, and default values for the outputs. It also implies a priority value for statements, so can be used to generate if-then-else like constructs. In essence, it allows almost arbitrary logic functions, as it allows the specification of all the inputs to the

function, the outputs, and the mappings between them. The nomenclature used is that the inputs are "Conditions", the outputs are "Actions", and the mapping between them is a "Conditional Action List". For each Conditional Action, any conditions not specified are assumed to be "don't cares", or conditions can be specified partially as don't care by supplying a mask value. For each Conditional Action, any actions not specified take their default value. The number of conditional actions can be arbitrary, and they are prioritized by order so the highest order ("last one on the list") that matches takes precedent. This allows the construction of 'if-then-else' type logic, and means that only one line in the TCAM will match. Any number of the output signals can be assigned values based on the line which matches. The one disadvantage of the TCAM is that for sets of conditional actions that may differ only in a single condition or single action, they must all be fully specified. An intelligent compiler could optimize this (for example, by assigning values to opcodes such that don't-care bits can be utilized efficiently).

The parameters for the TCAM are:

- $INPUT_LIST = [\{name => ..., width => ...\}]$
- $OUTPUT_LIST = [\{name => ..., width => ..., default => ...\}]$
- $CONDITIONAL_ACTION_LIST = [\{condition_list => [...], action_list => [...]\}]$

The INPUT_LIST parameter gives the names and widths of the input signals. The OUTPUT_LIST parameter gives the names, widths, and default values of the output signals. CONDITIONAL_ACTION_LIST gives the mappings from inputs to outputs. Each CONDITIONAL_ACTION_LIST array element corresponds to a line in the TCAM. The condition_list can contain for each input a value and compare mask. If an input is not in the condition_list, its compare mask is all don't-cares). The action_list gives the list of outputs and their values corresponding to a match of the condition_list. Any outputs not specified in the action_list take their default values.

Having defined the mapping, we use a TCAM (Figure 3.8, Figure 3.5③) to map inputs to events[2]. We use a second TCAM to map the current state and the triggered event to the next state and actions to perform (which FSM arc to traverse). Figure 3.9 shows the Transition TCAM in more detail, and its place in the generalized controller is shown in Figure 3.5④. The structures, though they use the same TCAM primitive, differ in that the

[2] This mapping could be dense or sparse, but we use a TCAM to make it more broadly applicable.

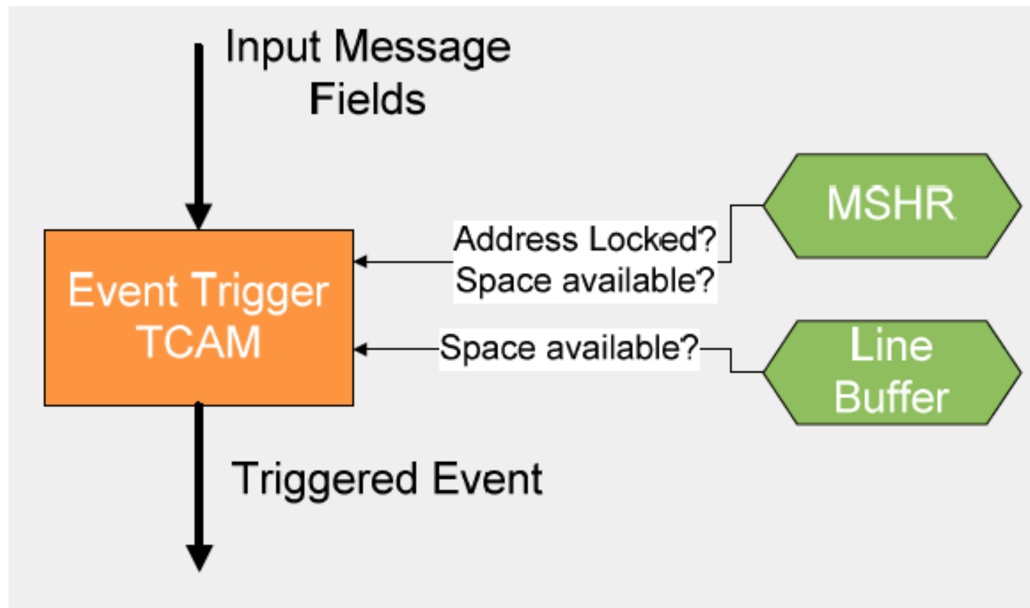


Figure 3.8: The Event Trigger TCAM, used to map fields from incoming messages and other blocks in the system to a single triggered event.

event trigger TCAM outputs only a single numerical event as its action, while the transition TCAM outputs a number of one-hot action signals and the numerical next state.

Once the protocol controller has used the Transition TCAM to determine which arc to traverse on the protocol, it needs to actually perform those actions. While up to this point, the structures used to construct the protocol controller are very generic and reusable, the actions could be "anything", so a variety of hardware might be needed to perform them. We introduce a translation block to implement more flexible logic.

3.4.3 Translations

One common action that is easy to abstract is the sending of messages out to other blocks. However, populating the fields of such a message could require arbitrary logic. For this reason, we introduce the concept of a translation. A translation is a way to program fairly arbitrary logic, but in a way that makes it cleanly translate from all its inputs to its outputs. The internal microblocks used to implement that logic can be selected from a variety of types. Rather than requiring that the outputs map to the inputs via only a single TCAM, LUT, or similar structure, the translation allows the programming tool to select from and

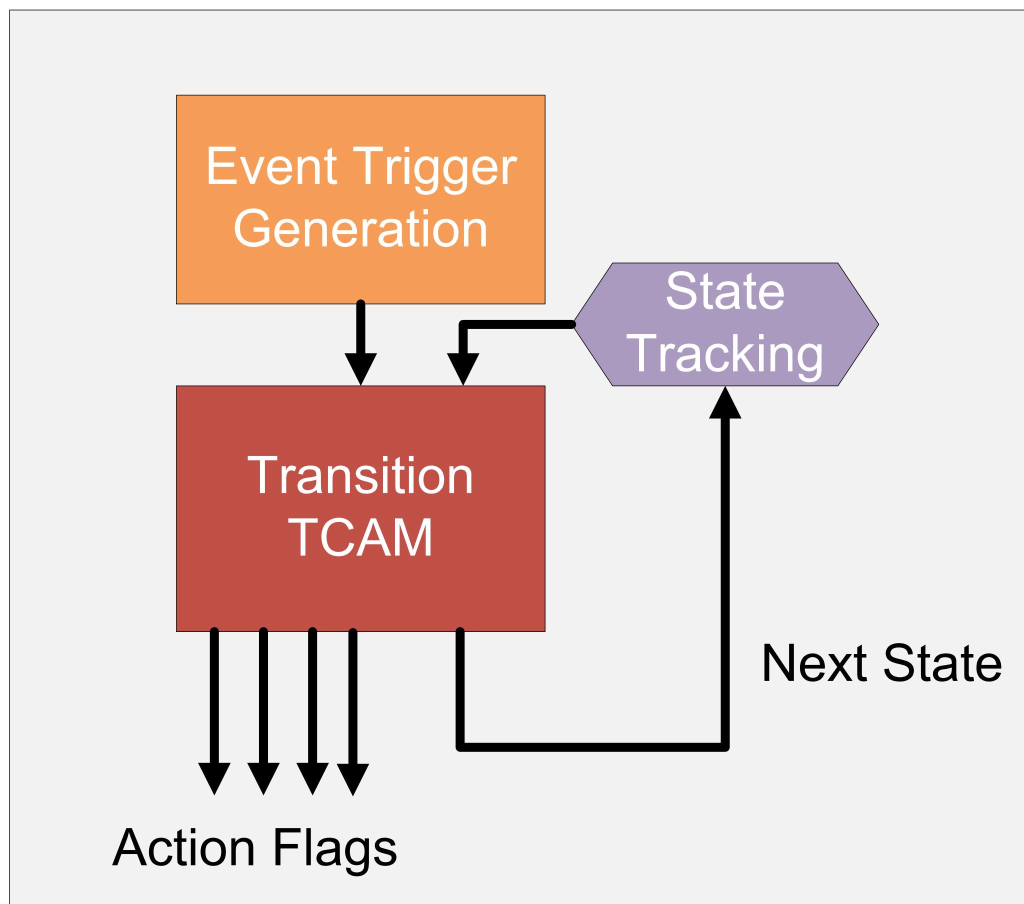


Figure 3.9: The Transition TCAM: used to generate the next state and actions to take given an input event and current state.

instantiate a variety of hardware structures in order to populate an outgoing message. It is referred to as a translation because it exists to generically translate inputs into outputs. The inputs and outputs are specified using interfaces, but these internal interfaces don't need to map exactly onto an interface between separate controller hardware. The generator assembles the interfaces, pulling in signals which need to be incorporated into the translation (logic) for the output interface signals. The translation block can generate either control or data signals which connect through an interface at the output of a translation block.

In order to populate an outgoing message, signals may be provided by a variety of methods. A simple one might be `immediate`, which would assign a constant value to any valid message (a useful optimization if all outgoing messages assign the same value). Another functionality would be to assign a field to the outgoing message by tying it to an input, for example the Address or sequence number of the outgoing message could be provided directly and consistently from another part of the system. For that the hardware could directly wire an output to an input using a `pass` or `rename` strategy. However, the logic could be more interesting. As an example, a `REFILL` request to off-chip memory may need to include the number of bytes required to complete the REFILL (because memory blocks use different line sizes). So, some mapping from requestor to that field in the output message is needed, and in this case a LUT is a good candidate. For more complex situations like the outgoing message type and validity, which depends on the actions needing to be performed, a TCAM can be instantiated.

While translations can be arbitrary logic, some types occur often enough that we include internal templates / generators for it. One such type is a TCAM mux, which uses the control inputs to select which data input to send to the output. While a TCAM allows a mapping from inputs to outputs, it does not perform well when one of the outputs should simply be equal to one of the inputs, because that would require fully populating all of the possible values of the inputs. For example, the protocol controller may be tracking the source of data to write to the off-chip memory system. In some cases it should be copied from the line buffer, while in others it might be copied directly from an incoming message to be forwarded directly out of the system. To provide this functionality, the TCAM mux uses a parameterized mux which is controlled by a specialized TCAM. The TCAM structure is controlled by the same idea of a set of conditions, but the only output is which selection to make. A separate set of inputs provides the possible selections to choose from. If some inputs should be constants, they can simply be provided to the TCAM mux as possible

inputs.

The parameters for the TCM Mux are :

- $INPUT_LIST = [\{name => ..., width => ...\}]$
- $DATA_LIST = [signal, ...]$
- $DATA_WIDTH = width$
- $CONDITIONAL_SELECT_LIST = [\{condition_list = [...], selection => signal\}]$

The INPUT_LIST parameter is essentially the same as the INPUT_LIST parameter for the TCAM. The DATA_LIST parameter provides the possible inputs that would be selected between, and the DATA_WIDTH parameter describes the width of those signals. It is required that all possible input signals have the same width, which will be the same as the output signal's width (there is no parameter to specify that as it uses the DATA_WIDTH parameter as well). The CONDITIONAL_SELECT_LIST parameter is similar to the TCAM's CONDITIONAL_ACTION_LIST, except instead of the Action list, there is only a single action, the selection, and is the name of the signal from DATA_LIST which should be selected when those conditions are true. Since the TCAM is the underlying structure, the TCAM Mux also allows a priority to its outputs.

One final way the translation can map an input into an output is to decode it into one-hot signals. This is useful for translating a destination id into the one-hot Valid signals on the outgoing interfaces.

Putting it all together, the translation block shown in Figure 3.11 provides a way for the hardware generator to select from a variety of hardware primitives rather than just program a single set block. The parameters for the Translation are:

- $IFC_IN = interfaceObject$
- $IFC_OUT = interfaceObject$
- $MAPPING_TYPES = [\{out_sig => ..., map_type => ...\}]$

The IFC_IN object demonstrates the power of the interface primitive, by wrapping up information about the input signals and their widths. The IFC_OUT object does the same for the output signals. The MAPPING_TYPES parameter describes how each output signal should be determined. If a signal from IFC_OUT is not found in the MAPPING_TYPES

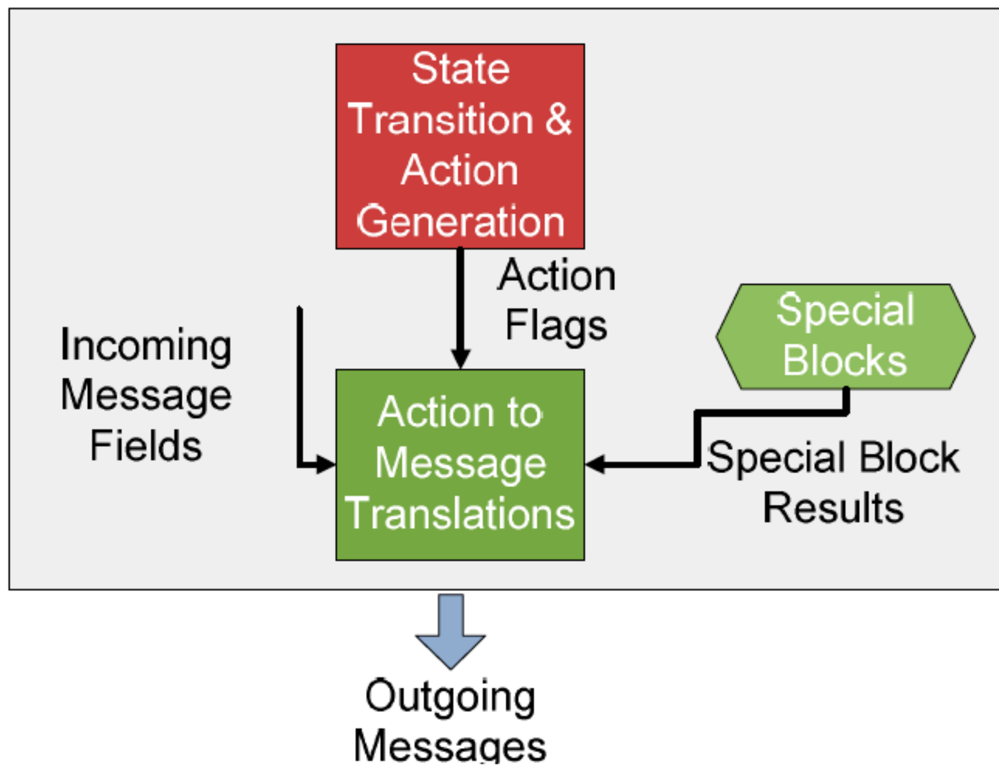


Figure 3.10: Action to Message Translation System, which converts action one-hot signals and a variety of data inputs into output messages on an interface. The Translation internals are shown in Figure 3.11.

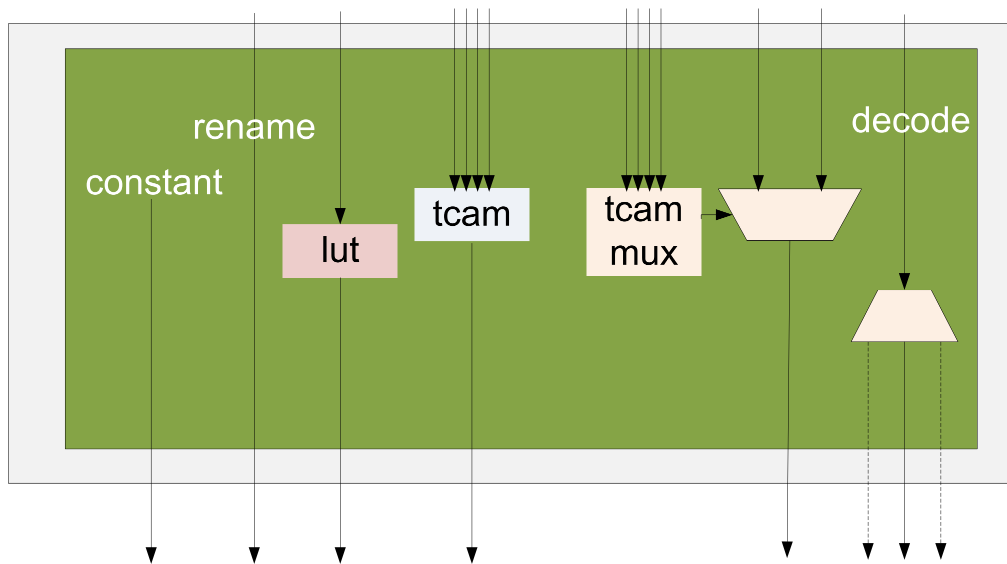


Figure 3.11: The Translation Primitive. This primitive maps actions and input data fields into outgoing messages, using a variety of mapping techniques. For each output, the mapping type is specified using a parameter, then the primitive instantiates the correct internal hardware for the mapping. Further parameterization indicates the functionality of the smaller internal blocks, such as the fields of a LUT or TCAM. The figure shows one mapping of each type, but an actual translation could have any number of each type of mapping.

array, then its mapping is assumed to be tied to zero. Otherwise, it can be explicitly assigned using one of the variety of mappings:

- **zero** The default, tie the output to zero.
- **pass** This requires that there is a signal in IFC_IN with the same name as IFC_OUT. The output signal is simply assigned to the input signal with the same name (its value is passed). The signals do not have to have the same width, the output signal will be truncated or padded with zeros accordingly.
- **rename**(in_sig = *signal*) Take an input signal and rename it to the output signal (useful for situations like where 'data' might get renamed to 'returncode'). If the output signal is not the same width as the input signal, it will be truncated or padded with zeroes accordingly.
- **immediate**(val = *value*) Give an immediate value (such as 1 or 0) to a signal. This is fixed and useful for debugging.
- **decode** The translation will instantiate a decoder and decode the value of the signal. This is handy for converting a binary signal into a one-hot value.[3]
- **lut**(lut_param = *signal*, [lut_module = *module*]) This requires the additional information of a signal to act as the input to a LUT. The Translation will instantiate a LUT, where the input to the LUT is the signal specified as `lut_param`. The LUT configuration is parameterized internally, or an alternative is that an already instantiated LUT object be passed in. In that case, the LUT will be cloned an instantiated, thereby using the same internal parameterization as the LUT passed in as `lut_module`.
- **tcam** (tcam_params = *[signals]*, [tcam_module = *module*) This instantiates a TCAM for a given output signal. The multiple inputs which control the TCAM are listed in `tcam_params`. Similar to the LUT, the translation will instantiate a TCAM, which will have its CONDITIONAL_ACTION_LIST provided by its own internal configuration, or it can clone a tcam which is passed into the tcam_module parameter to duplicate its internal configuration.

[3] There is not currently an encoder primitive though intuition tells us that would be a good idea, it just never came up in our use cases.

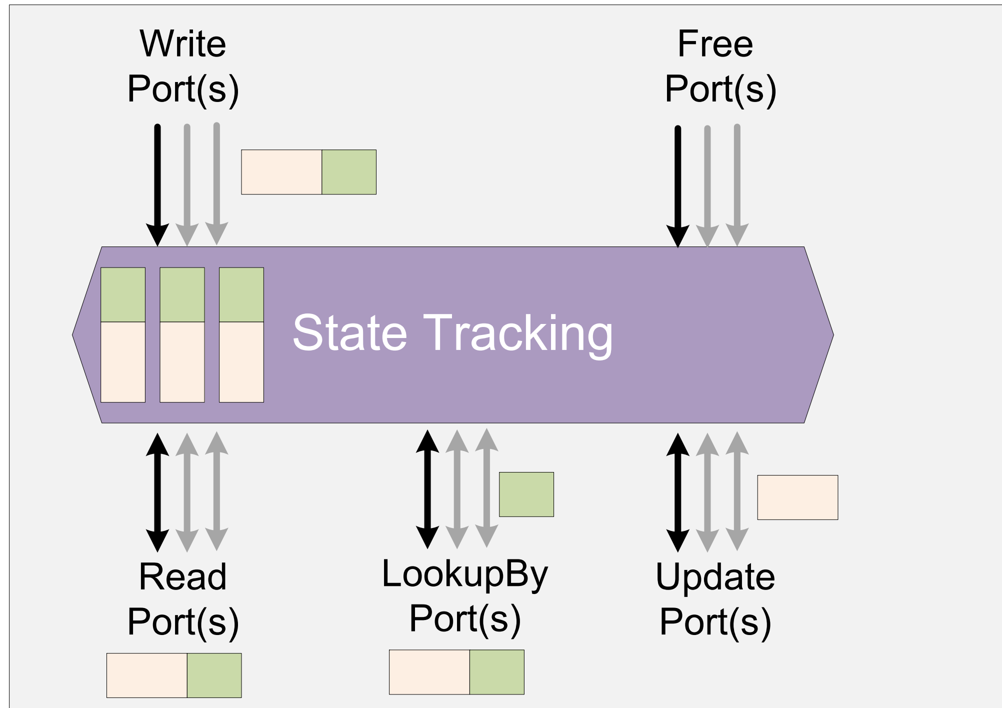


Figure 3.12: State Tracking Hardware, for tracking the state of multiple operations as they pass through the controller. The hardware can have any number of fields in a single entry, and can have any number of entries. One field in the entry is always 'State'. The write ports allow writing of an entire entry (identified by its index) at once, and read ports read the entire indexed entry at once. A LookupBy$< X >$ port allows looking up an entire entry by a field X, rather than by its index. Free ports deallocate an index when it is no longer needed. Update ports allow updating the individual fields of an entry (such as State) without modifying its other fields.

- **mux** (tcam_params = *[signals]*, mux_options = *[signals]*) The translation will internally instantiate a TCAM mux. The control signals to the TCAM mux are chosen from the input signals by listing them in `tcam_params`. The possible inputs are listed in `mux_options`.

3.4.4 State Tracking and Special Blocks

One of the key jobs of the protocol controller in implementing its portion of the memory protocol has been neglected up to this point. That is, the protocol controller needs to be able to track and update the current state of the system. The state transition diagram

in Figure 3.4 shows an abstract view of the state of a single line in the system. From the protocol controller's perspective, however, the actual state of a line is more fine-grained because it cannot atomically perform all the actions which are needed to make the transition valid. Therefore, the view of the protocol controller has many more than three states for even this simple example. In addition, the protocol controller needs to track multiple operations at a time, and have a way of associating or looking up the state whenever an event is triggered. For this, we introduce a *state tracking* primitive, shown in Figure 3.12. The State tracking primitive is used to build the tracking structures (eg, MSHR), and holds at a minimum the State of each outstanding transaction in the controller. It also holds any associated information needed to identify the transaction (such as Address or originating processor) and can hold additional data (such as an index in the Line Buffer which is accumulating the data for this request).

The parameters for the State primitive are:

- $NUM_ENTRIES = number$
- $STATE_IFC = interfaceObject$
- $NUM_READS = number$
- $NUM_LOOKUPS = number$
- $LOOKUP_SIGNALS = [signal, ...]$
- $NUM_WRITES = number$
- $NUM_FREES = number$

The NUM_ENTRIES parameter controls how many entries are available to track in the State. STATE_IFC parameter provides all the information about the signals contained in each entry of the State. NUM_READS, NUM_LOOKUPS, NUM_WRITES, and NUM_FREES indicate how many ports of each type should be allowed. Conflicts in priority are handled externally (an index that is freed/allocated/written/updated at the same time is an error, and writes/updates are prioritized by the port number). The LOOKUP_SIGNALS parameter lists which signals should be included in the lookup ports. For example, if LOOKUP_SIGNALS = ['Address'], then there will be an input 'LookupBy_Address_0' (and multiple of these if NUM_LOOKUPS > 1). The state tracking primitive thereby provides

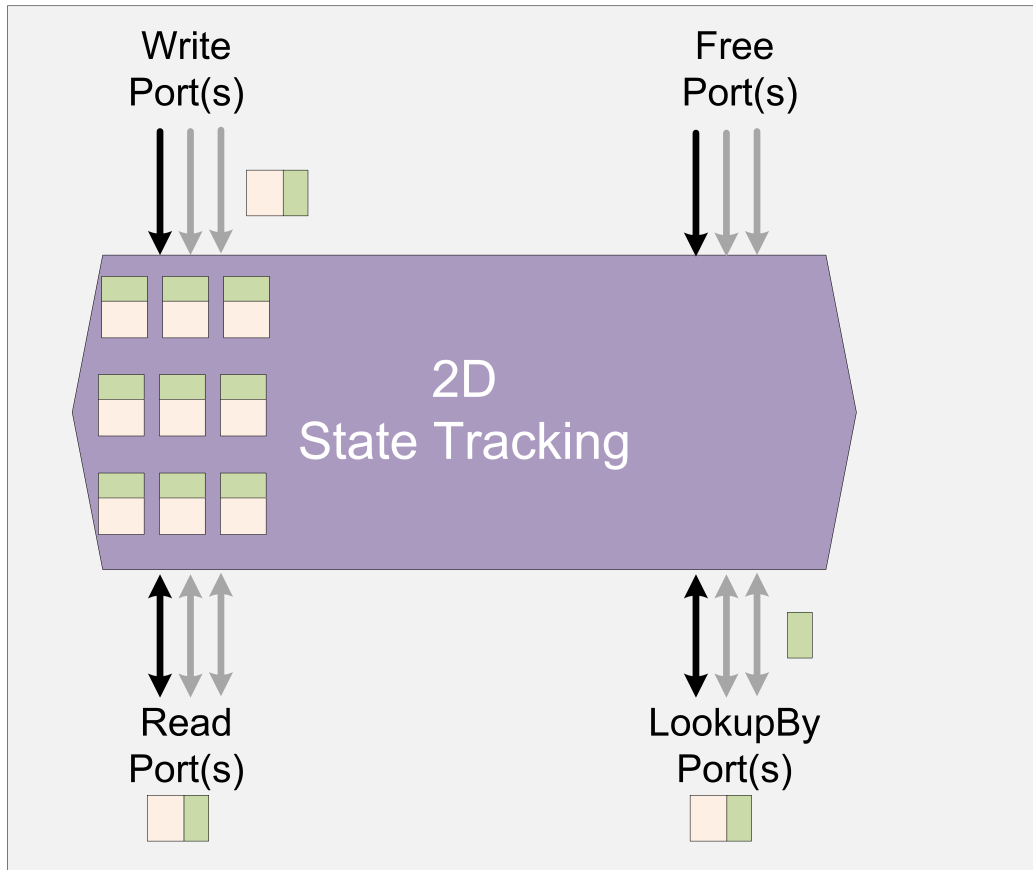


Figure 3.13: Two Dimensional State Tracking Hardware, for tracking the state of multiple operations as they pass through the controller, by allowing for an Offset field in addition to a basic "LookupBy" field. This allows multiple items (pink squares) of specified sorts to be stored with the indexing items (green squares), and to update them in a smaller granularity.

an understandable way of creating a small memory with entries that can be read, written, or updated in a single cycle.

In some cases, the data that the protocol controller would like to store with the state is too large to make updating it in a single cycle impractical. For example, when writing a cache line into the line buffer, it could require several cycles to receive all the data from the off-chip network. Therefore, a *two dimensional state* primitive (Figure 3.13) extends the regular state primitive into something that looks like a two dimensional memory. You can now specify a number of entries in a horizontal direction as well as a vertical direction. There is no lookup capability in the 2D State, but you can still have multiple reads, writes,

and frees. Reads and writes now specify an offset and only the entry at that offset is read or written. A line in the state is Valid once one entry in the line is Written, and it goes invalid once that line is explicitly freed. A good example is a LineBuffer structure, where it is impractical to read and write the entire entry at once, and one would rather do it on a word basis rather than a line basis.

In order to populate outgoing messages with data from the 2D structure (for example, sending a line of data into a memory block), the generator for the Protocol Controller can choose to instantiate a special *message pipe* primitive whenever it has to send some of these "wide" data fields, which breaks wide signals into narrow, more practical ones. An example is converting a single abstract message directing the memory to send an entire cache line to a cache (e.g. on a refill). The message pipe (shown in Figure 3.14) is similar to the translation in structure with all fields as "pass", but it contains state in order to track the messages flowing through it. It can be inserted between the translation block and the outgoing interface (between ⑧ and ⑨ in Figure 3.5).

The message pipe saves the short, abstract command (e.g. "REFILL"), then outputs several true messages with an associated offset to complete the full abstract operation. It increments the offset on each of the true messages, and the number of messages needed to complete the operation is provided by a lookup table indexed by the Type of the incoming message. As an example, a message pipe would could be instantiated between the protocol controller and the memory block (cache). Abstractly, the protocol controller could send a single data write message containing an entire new cache line to refill the cache. The message pipe intercepts that message, and handles sending several smaller data write messages to the cache, incrementing an offset counter both to read from the line buffer and to indicate the offset to the cache itself. This functionality is hidden from the user.

The parameters for the message pipe are:

- $IFC_IN = interfaceObject$
- $PIPED_SIGNALS = [signal, ...]$
- $LUT_PARAMS = [...]$
- $COUNT_WIDTH = number$

The IFC_IN parameter describes both the input and output interface for this message. The PIPED_SIGNALS parameter lists the signals which need to be pipelined (the other signals

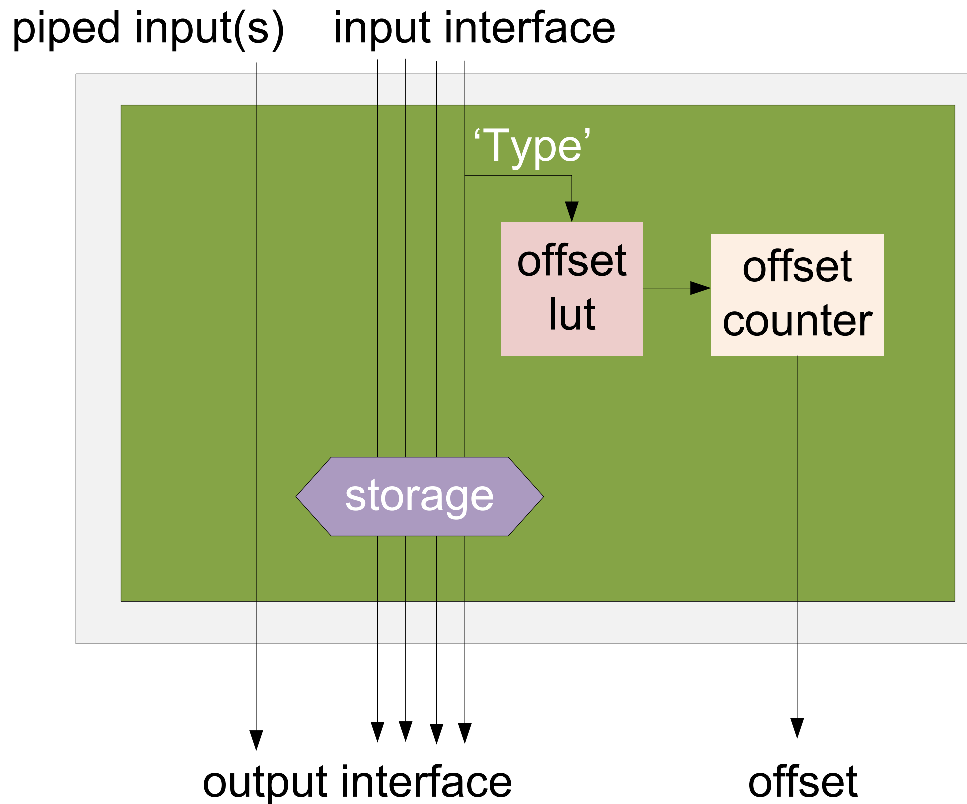


Figure 3.14: Message Pipe Hardware, for allowing some wide fields of messages to be "piped" over multiple cycles. When the input message is received on the interface, its Type field is used to look up the total number of messages that it will need to be broken into, and (if that is greater than 1) the offset counter is initialized, and the input message is saved. On each cycle the offset counter is sent to both the receiving block and to the two-dimensional structure holding the wide data. The value read from the two-dimensional structure is received on the "piped input", and incorporated into the message going on the output interface along with the stored fields. When the counter decrements to zero, the message pipe resets to the idle state.

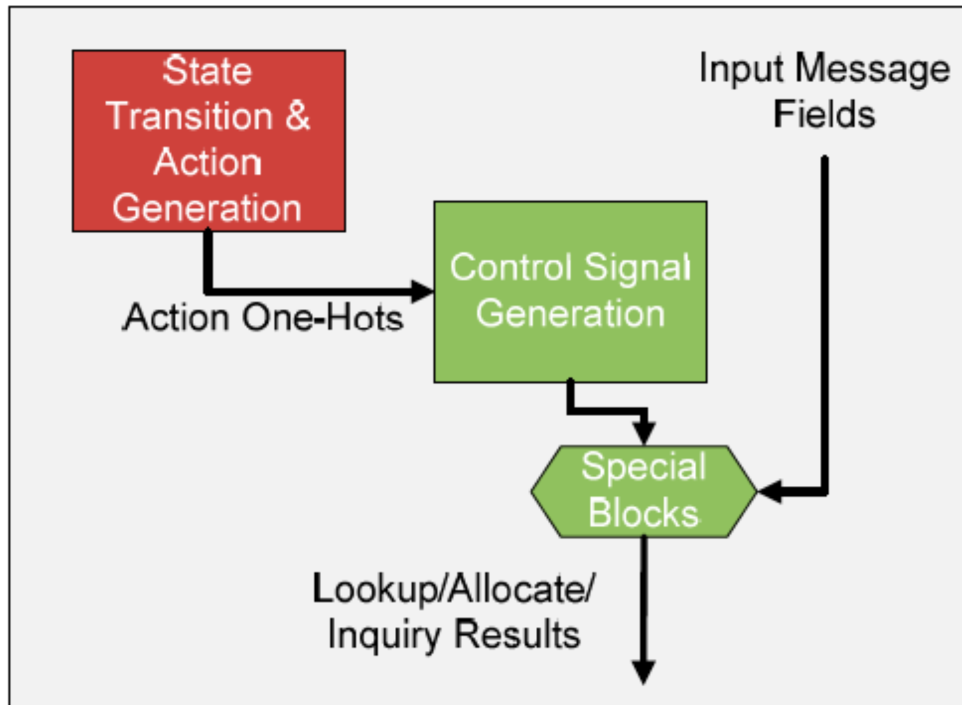


Figure 3.15: Hardware for generating control signals and inputs for the Special Blocks.

in IFC_IN will just be saved and be the same for each output message). The LUT_PARAMS enables the Message Pipe to instantiate a lookup table and use it to look up the pipeline depth for each 'Type' coming on IFC_IN. The COUNT_WIDTH just gives an upper bound on the size of counter necessary to implement the maximum pipeline depth.

Finally, if the protocol controller has any state which it knows will be completed in order (in our system's case, the protocol controller does not have this restriction, but the reply handler does), an in-order FIFO State can provide a more efficient tracking structure. The FIFO state is very similar to the State, but it allows multi-ported FIFO-style reads and writes rather than indexed reads and writes. This is useful for state entry trackers which are allocated and deallocated in order.

Despite all these programmable hardware primitives, in order to fully implement a protocol, the protocol controller and other blocks in the system will need some structures and functions which are not fully specified by the programming parameters. These *special blocks* and/or functions can be hardcoded, but in order to make the hardware flexible, they

each satisfy a clearly specified interface. One example is the block which returns the best eviction candidate in a cache when provided with a target address to find a home for. The internals of the block are written in Verilog, but different blocks with the same interface could be swapped using a parameter to select which to instantiate (this is a classic use of Genesis). The control signals for these blocks face the same problems as sending messages to other entities in the systems (in fact, one could view their interfaces as internal messaging systems which need to be controlled in the same way). Figure 3.15 shows how actions can be interpreted in order to provide inputs to the special blocks. Note the similarity to Figure 3.10. By using the same paramterizable interface structure, we can build the control logic for the special functional units in the same way as for sending messages to other blocks.

The state tracking primitives can be viewed as special cases of the special blocks, in that they are highly specialized logic blocks that we know how to build. A useful approach for special blocks in general is to create a parameterized Verilog module that can be reused for different purposes, and has a clearly defined interface that can be used by the compiler.

This chapter has described a memory system architecture composed of controllers containing a large number of small, simple building blocks. All of these pieces can be configured to implement the types of memory protocols similar to the one shown in Figure 3.4. The next challenge is to make the problem of configuring all the parameters in these blocks surmountable, by moving the specification to a higher level. The following chapter describes our specification language to make this a tenable task.

Chapter 4

Programming the Hardware: SLAMM

The previous chapter described how to build one of the controllers needed to implement a memory protocol, and explained the flexible hardware blocks that are contained within it which allow programmability of the system. By using parameters to specify the system at design time and only building the hardware particular to that system, we are no longer constrained by hardware that has to be manually connected between blocks, inputs to logic, etc. However, we have only partially addressed the problems we had in earlier configurable systems, because selecting the necessary values for the many parameters in the system is a Sisyphean task. Therefore, this section describes a higher level language, and compiler flow, that we have developed to make programming the system a more natural task for a protocol developer. This language, Specification Language for Advanced Memory Modules (SLAMM) provides a programming capability that is similar to C, but with strong ties to the understood architectural model for which it is being programmed. Section 4.1 explains the grammar and constructs of this language in detail, and runs through an example of specifying a thread of control in that language. Section 4.2 explains how the compiler for SLAMM integrates the parameters from the high level specification into the format needed by our controller template. Section 4.3 gives a quick summary of the ways our current templates use (or do not use) this high level input, to demonstrate how even a partial specification with this language can be useful, and be integrated with already coded, relatively inflexible blocks. Section 4.4 then discusses an example of extending a traditional cache model with this specification language.

4.1 The SLAMM Constructs

SLAMM preserves the idea that we developed in the previous chapter, that in order to make our system efficient and have a lot of parallelism, we should have a lot of smaller controllers working independently, sending messages to each other. Therefore, the highest level construct in the SLAMM language is the *state machine*, and a configuration program contains one or more state machine descriptions, indicated with the `machine` keyword. The state machine description describes the connections to other blocks in the system, and contains all the information that would be in the controller's internal state transition diagram (a more complicated version of that shown in Figure 3.4). Each state machine description has to describe the states that make up its internal protocol, events that it recognizes, and actions that it can perform. It also needs to describe the transitions which link those states, events, and actions. Again, while the states that the controller is tracking is a property of the state machine, each of these (states, events, and transitions) are actually occurring on a 'line' or Address granularity. So, as far as the description is concerned, the specification is generally for a given line. For example, a MESI protocol would refer to a 'line in the M state', not a 'cache in the M state'. Figure 4.2 shows the basic template for a state machine code specification.

Figure 4.1 shows an example flow through our abstract controller, which we will revisit throughout this chapter in order to show how the SLAMM language describes what should happen in the system. The scope of a state machine file is the scope of the single controller in Figure 4.1. In order to fully specify the behavior of a hierarchical, interconnected system like the one in Figure 3.3, the user needs to write several such state machine files, then the SLAMM program is composed of all of them, plus additional header file information. The first and most important class of information in header files is the message descriptions.

4.1.1 Messages

As shown in the overall system architecture in Figure 3.6, we don't get very far describing only the internals of each state machine blocks, we especially need to specify and configure the interfaces between them. What the hardware requires is a description of the signals between each pair of blocks and their widths (or if those are not known, a pointer to where to get the correct width). Therefore, a large part of the SLAMM language is devoted to describing the interfaces between controllers, called Messages. These are declared separately

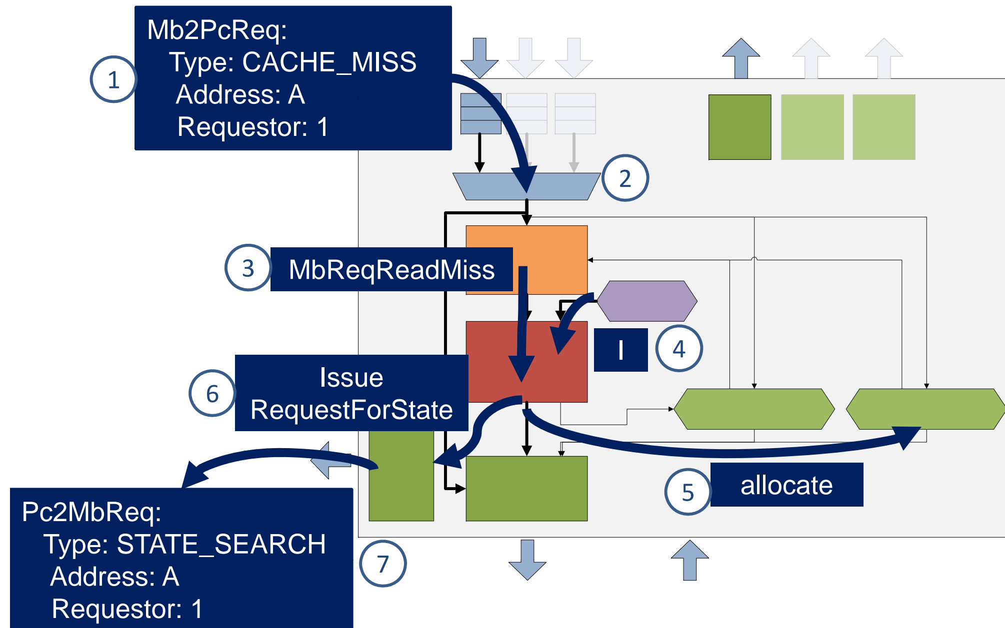


Figure 4.1: Flow through the Protocol Controller as specified by SLAMM. 1) The Protocol Controller receives a valid Memory Block Request message on one of the interfaces, of type CACHE_MISS to address A. 2) The request is buffered and then arbitrated in structures specially configured for holding requests of that type. 3) The request is examined by the Event Trigger TCAM, and because of its message type an event of type MbReqReadMiss is generated. 4) The current state of the line is looked up and seen to be Invalid. 5) By combining the event and the current state, actions are generated by the Transition TCAM. One action is to send a special `allocate` command to the MSHR special block. 6) At the same time, another action is to issue RequestForState message. 7) The Output Message Translation block turns the action into an output message, and also pulls in the Address and Requestor signals from the original message.

```
machine(ProtocolController, "MESI Protocol Controller (PC)") {

  // Enumeration of STATES

  // Enumeration of EVENTS

  // Enumeration of internal messages (eg to MSHR)
  // Enumeration of internal state structures

  // Declaration of EXTERNAL (‘‘Special’’) BLOCKS/INTERFACES

  // Primitives for STATE QUERY/UPDATE (not currently used by SLAMM!)

  // OUTPUT PORT declarations

  // INPUT PORT declarations and TRIGGER generation

  // ACTION Descriptions

  // TRANSITIONS (map from current state and trigger to next state and actions)

}
```

Figure 4.2: Code for a State Machine. A full SLAMM specification is composed of several such blocks of code, and additional code to describe the messages between them.

```
enumeration(Mb2PcReqType, desc="...") {
  READ_MISS,      desc="Read Miss (line was not present in cache)";
  WRITE_MISS,     desc="Write Miss (line was not present in cache)";
  UPGRADE_MISS,   desc="Upgrade Miss (line was present but not exclusive)";
}


structure(Mb2PcReqMsg, desc="...", interface="Message") {
  Mb2PcReqType Opcode,   desc="Type of request (CACHE_MISS, etc)";
  Address Address,             desc="Address for this request";
  DataBlock Data,           desc="data for the cache line";
}

//SLAMM looks for ports with specific names,
// like mb2cpu_rep, which match the architectural model
out_port(mb2cpu_rep, Mb2CpuRepMsg, mb2cpu);

in_port(cpu2mb_req, Cpu2MbReqMsg, cpu2mb, desc="Reqs from CPU to Cache"){
//EVENT GENERATIONS (See Below)
...
}
```

Figure 4.3: Code for Messages, which help configure the interfaces between controllers, as well as interfaces between internal controller components. The message enumerations and structures are described separately from an individual state machine so that they can be shared, while the port declarations are coded within a specific state machine.

for every physical interface in the system, using the `structure` keyword, but with a special name which ends in 'Msg'. For example, the interface from Protocol Controller back to the cache is described with the Pc2MbReqMsg. Each *Msg structure has an associated MessageType, which is a special case of the `enumeration` keyword.

In the previous example, Pc2MbReqMsg structure will have a Pc2MbReqType field. The fields of the message structure define a given physical interface. While the width of some signals on the interface are deduced by SLAMM (for example, the number of bits needed to describe the message Type field), the other widths come from the type of data stored in the field. Many of these widths are unknown and intentionally ignored by SLAMM.

The SLAMM compiler knows how to translate data types such as DataBlock and Address into 'late bound' parameters, which are defined later by the hardware. In this way, SLAMM does not need to be concerned with the width of signals, and the same specification can be applied to an instruction cache with 64-bit instruction words and a data cache with 32-bit data words. For example, SLAMM will associate a message field variable with type 'Address' with a width LATE_BIND_ADDRESS_WIDTH, then the hardware will find the correct ADDRESS_WIDTH parameter (by default, in the parent module), and throw an error if it doesn't exist. This helps keep SLAMM separate from the implementation, as the protocol designer is not usually concerned about the value of these widths, but they must be known when compiling real hardware.

Figure 4.3 shows the templates for the parts of the language associated with messages. While Messages and Message Types are not necessarily affiliated with a given state machine (their whole purpose is to connect up different state machines), Message Queues are defined within a state machine as input and output ports, using the `in_port` and `out_port` keywords, respectively. For a given architecture (for example, the hierarchical memory system shown in Figure 3.2), the compiler actually looks for specific *Msg structures that describe the interfaces. It also looks for the specific in_port declaration in order to define event triggers, and for out_port declarations in order to populate outgoing messages. A different architecture with additional levels of hierarchy would require additional SLAMM state machine specifications for the new controllers at the new level, and the compiler would be modified to know to look for the state machine specifications for controllers at the additional levels, and for the interfaces between them.

Another important aspect of the message definitions is that SLAMM provides an abstraction that does not need to be obeyed literally by the hardware. Specifically, signal

```
// STATES
 enumeration(MbLineState, desc="Cache states") {
   //Base States
   I, desc="Not Present/Invalid";
   S, desc="Shared";
   E, desc="Exclusive";
   M,  desc="Modified";

   R, desc="Reserved";
   T, desc="Transient";
 }
```

Figure 4.4: Code for States

widths do not need to be maintained if there is a marshalling/unmarshalling hardware for wide interfaces. While the SLAMM specification may say "copy one cache line from Memory System's Reply into the line buffer", the hardware can transparently marshall and unmarshall such requests in order to make the connectivity consume fewer actual wires, using the message pipe structure we described in Section 3.4.

In our example shown in Figure 4.1, the user defines two message types, `Mb2PcReq` and `Pc2MbReq`. For each type they define fields Type, Address, and Requestor. Not shown are the additional types for the replies and requests going out on other interfaces.

4.1.2 State and State Updates

Enumerating states of a "line" is simple, using the `enumeration` keyword with special type State. The language automatically associates this with the containing State Machine, so converts it to MemoryBlockState enumeration, for example. To specify the states, they are simply listed, the SLAMM language assigns and tracks a numerical value for each.

Currently, the user has to list all the states and the transient states, which might be implementation dependent. For example, if in the course of one state transition the controller needs to send a message and get a reply back to decide what your next "real" state is, the user has to specify a state for "message sent, waiting for reply". If replies from multiple different entities are required, (for example, waiting for the eviction state from the memory block and the read data from the off-chip memory) then intermediate states need to be constructed in order to capture this. This suboptimal behavior could be improved upon by

building a special block which handles only this tracking to provide a single input to the state tracking unit indicating when all replies have been received. Figure 4.4 shows how to enumerate the States of a line. These are specifically for the line states controlled via transitions (see below), other information is also used and tracked via an `external_type`, as described below.

It is notable that that the code the user has to write is not a one-to-one mapping from the State Machine diagram. The first step is for the user to identify the states that the line can be in inside the state machine of interest. In our example controller in Figure 4.1, the line is either not present (I= Invalid), or it is (A = Active). A finer division of these states is necessary as we handle the request. The user can use the SLAMM `State` construct to enumerate the states.

4.1.3 Control Flow

In order to provide the parameters needed to populate the Event Trigger TCAM discussed in the previous chapter, SLAMM uses `Events`. Events are described with the `enumeration` keyword with the special type `Event`, and are automatically associated with the State Machine that they are declared within.

To describe the actions which need to be performed and the next state when a given event occurs in the current state, SLAMM uses the `transition` keyword, which takes the initial state(s), and triggering event, and outputs the final state and actions to perform in order to complete the transition. If the final state is not specified it indicates to keep the state the same.

The actions which are to be performed are specified with the `action` keyword. As described in the previous chapter, one major class of actions is those that send messages to other controllers in the system via the output ports. These actions contain a description which uses the `enqueue` special function keyword. The enqueue function specifies a destination output port and describes how to populate the outgoing message fields. The `peek` special function keyword can be used to examine the message which triggered this action, in order to copy fields from it or use them to look up other information throughout the system. The SLAMM compiler must examine all possible sources of data for each field of an output message interface. For each field, it selects the proper type of hardware structure to instantiate inside that output message translation, and makes sure that any necessary inputs are provided to the output message translation. Other actions, such as allocating or

```
// EVENTS
enumeration(Event, desc="Memory Block Events") {
  // From processor
  Load,       desc="Load request from processor";
  Store,      desc="Store request from processor";

  //from PC (Search Access Gen)
  StateUpdateChangeTagToM, desc = "Change tag to Modified";
  ...
  //from PC (Data Pipe Acces Gen)
  DataAccessRead, desc = "Read out data from a specific way";
  ...
  //from PC (Responses)
  EventCompletion, desc = "Event Completion response from the PC";
}

//TRIGGERS GENERATED FROM MESSAGES RECEIVED
in_port(cpu2mb_req, Cpu2MbReqMsg, cpu2mb,
        desc="Reqs from CPU to Memory Block") {
  peek(cpu2mb_req, Cpu2MbReqMsg) { // REquired...
      if (in_msg.Type == CpuReqType:LD) {
        trigger(Event:Load, in_msg.Address);
      } else ...
      //Other Events triggered based on in_msg fields
  }
}

// ACTIONS
action(l_load_hit, "l", desc="Return data to the requesting processor") {
  enqueue(mb2cpu_rep, Mb2CpuRepMsg){
    out_msg.Data := cacheMemoryBlock[address].DataBlk[address];
    out_msg.Type := Mb2CpuRepType:HIT;
  }
}

// TRANSITIONS
transition({M,S,E}, Load) {
  t_updateMru;
  l_load_hit;
  c_popCpu2MbReqQueue;
}
```

Figure 4.5: Code for Controlling the Flow of Activity. Each of these is coded within a state machine.

deallocating entries in the state tracking structures, don't need a description of what they do, because their action signals are used directly. In the SLAMM specification, actions have a name (which maps directly to a wire in hardware) and a description of what they do.

The list of actions assumes atomicity: there is no meaning for something that happens *while* those actions are being performed. Therefore, the hardware must enforce arbitration such that only one such thing happens at a time (or provide pipelining primitives "behind the scenes"). The pipeline insertion is beyond the scope of the SLAMM specification. The message pipe division mentioned in the previous chapter is also orthogonal to the SLAMM specification (the template may enforce that certain interfaces have a maximum width). Figure 4.5 shows how to enumerate Events, describe Actions, and describe the Transitions which hook everything together.

In our example in Figure 4.1, if the user's controller receives a CACHE_MISS and the line is in the 'I' state, the protocol controller would issue a request back to the cache to double check the state of the line. At the same time, it might inquire about a suitable eviction candidate and get the data currently at the line if it is actually valid. Therefore the user identifies an IssueRequestForState `action` and builds up the request with the `enqueue` language construct. The protocol also updates the state from 'I' to 'A', but since there are going to be multiple substates in "Active", the user constructs some intermediate states. In this case they create 'AS', as in "Active, waiting for State information". Other subsets of the Active state could be 'AM' or 'AWD', "Active, waiting for Memory System" or "Active, writing data" respectively.

When the response from the Memory Block is received, the `input_port` construct again compares against fields in the message. This time the comparison is more complicated, to compare both the message type (REPLY_WITH_STATE) and a field within the message, `LineState`. Based on the value of the line state, we might issue a variety of `event`s, such as `stateReturnedActuallyValid` or `stateReturnedEvictionValid` or `stateReturnedNoneValid`. If it was the case that the event was actually stateReturnedNoneValid, the controller needs to issue a request to the network for data, and also reserve a space for the new line in the Memory Block. Therefore, two actions issue in parallel, one to the Main Memory with a request for a line of data, and one to the memory block to reserve the line. There is no guarantee or requirement concerning which of these will complete first, so the transient state transitions are described in such a way that both are completed before moving on.

```
external_type(MshrTable){

//Commands
MshrEntry allocate(Address);
void freeByIndex(int);

//Inquiries
bool isAddressPresent(Address);
...
DataBlock getData(Address);
WayNum getWay(Address);
Address getAddress();
}
```

Figure 4.6: Example Code for “Special” (Externally Defined) Blocks

Once both actions have completed, the protocol controller can then copy the data that is now in the line buffer into a message for the memory block. Again there is the problem that the message is much wider than the physical interface, so unbeknownst to the SLAMM programmer, the `message_pipe` hardware is used to automatically read from the line buffer, incrementing the offset counter and sending it along to the Memory Block, which also understands offset. Because the offset is not zero, the state is not updated until the last data write is completed.

At that point, the Protocol Controller updates the Tag by sending another message to the Memory Block. Once that is acknowledged it completes the operation by sending the critical word on the Protocol Controller to Memory Block Reply interface, which gets back to the reply handler so it can unstall the processor and free its outstanding operation state.

4.1.4 Special Blocks

SLAMM cannot possibly describe all the capabilities of the hardware, and does not aim to do so. The hardware expects and allows additional or replacement blocks for certain functionality, such as a more sophisticated state tracking device, or a function for determining the eviction candidate. SLAMM provides a method for defining blocks without defined internal functionality, using the `external_type` keyword. Certain blocks, such as the Line Buffer and Mshr Table, may be assumed to exist in a given template, but the user could

Table 4.1: Using SLAMM To Provide Memory System Primitives

Required Language Primitive	**SLAMM Construct**
Controllers (FSMs)	State Machines
Interfaces between Actors	Messages, Input/Output Ports
Micro-code Inputs	Events(RX messages)
States	States
Transitions/State updates	Transitions w/ primitives for state update
Actions	Actions/External Actions
Conditionals	Conditionally Generated Events
Special Modules, Internal State Tracking	External Types
Special Functions	External Types

create a new external type as long as there was a corresponding Verilog module. These blocks can contain methods, and complex data types can be declared using the `structure` keyword. For our example in Figure 4.1, the MshrEntry structure can be defined with various fields, then the MshrTable can have a method `MshrEntry getEntry(Address)`, which SLAMM will be able to translate into a hardware interface. In fact, the get* method is a special case which SLAMM can translate into a look up all or part of the field stored in the table. Specialized methods such as getting the eviction candidate can also be described with this construct. Figure 4.6 shows how to define an interface to a special block.

The `external_type` construct can also be used to define special data types, such as WayNum, or DataBlock. This requires that the compiler be provided the mapping from this data type into the correct name for including with the LATE_BIND method. For example, the compiler can contain a mapping for the external_type WayNum should map to a width of LATE_BIND_WAY_SIZE.

Table 4.1 gives a quick overview of how the hardware components described in the previous chapter get their configuration values from the SLAMM constructs. Figure 4.7 shows the grammar used by SLAMM to describe the implemented constructs. This is a summarized grammar showing the more interesting aspects such as keywords, supported binary operations, etc. A full grammar is provided in Appendix B.

Once the user has described their protocol in this language, the SLAMM compiler converts the description to the parameters described in the previous chapter. The next section details this process.

```
file: decl_list
decl:  MACHINE_DECL     ( ident pair_list ) { decl_list }
    |  ACTION_DECL      ( ident pair_list ) statement_list
    |  IN_PORT_DECL     ( ident , type , var pair_list ) statement_list
    |  OUT_PORT_DECL    ( ident , type , var pair_list ) ;
    |  TRANSITION_DECL  ( ident_list , ident_list , ident pair_list ) ident_list
    |  TRANSITION_DECL  ( ident_list , ident_list         pair_list ) ident_list
    |  EXTERN_TYPE_DECL ( type pair_list ) ;
    |  EXTERN_TYPE_DECL ( type pair_list ) { type_methods }
    |  GLOBAL_DECL      ( type pair_list ) { type_members }
    |  STRUCT_DECL      ( type pair_list ) { type_members }
    |  ENUM_DECL        ( type pair_list ) { type_enums   }
    |  type ident pair_list ;
    |  type ident ( formal_param_list ) pair_list ;
    |  void ident ( formal_param_list ) pair_list ;
    |  type ident ( formal_param_list ) pair_list statement_list
    |  void ident ( formal_param_list ) pair_list statement_list
pair     : ident = STRING
         | ident = ident
         | STRING
statement:  expr ;
         |  expr ASSIGN expr ;
         |  ENQUEUE      ( var , type pair_list ) statement_list
         |  PEEK         ( var , type ) statement_list
         |  if_statement
         |  RETURN expr ;
if_statement: IF ( expr ) statement_list ELSE statement_list
            | IF ( expr ) statement_list
            | IF ( expr ) statement_list ELSE if_statement
expr:  var
    |  literal
    |  enumeration
    |  ident ( expr_list )
    |  expr . field
    |  expr . ident ( expr_list )
    |  type . ident ( expr_list )
    |  expr [ expr_list ]
    |  expr ==    expr
    |  expr !=    expr
    |  expr &&   expr
    |  expr ||    expr
    |  ( expr )
literal: STRING
       | NUMBER
       | FLOATNUMBER
       | LIT_BOOL
enumeration: ident : ident
var: ident
field: ident
```

Figure 4.7: The SLAMM Grammar, summarized to highlight keywords, supported binary operations, etc.

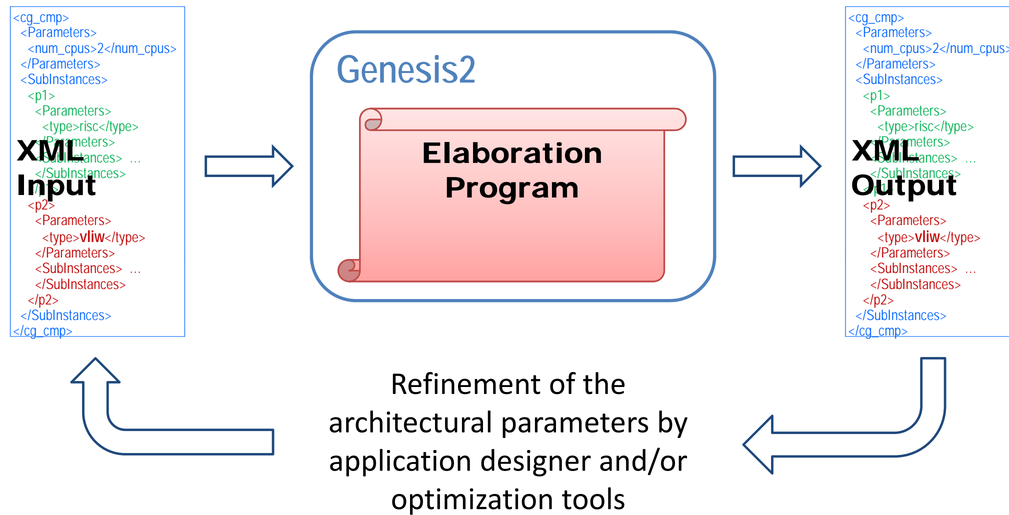


Figure 4.8: Illustration of the iterative process used by Genesis to customize designs by refining the XML description of the parameters' space. External tools can control the generated hardware by just modifying the XML structures (without touching any RTL descriptions).

4.2 Integrating SLAMM Parameters

In our implementation the hardware template described in Chapter 3 is written in Genesis [36, 35]. Figure 4.8 shows Genesis' parameterization loop. Genesis can take as input an XML configuration (or none, relying on default parameters), configure the system, then output an XML configuration file describing the system. The output XML (or the original configuration file) can be modified by external tools to create new configurations. Thus for this architecture, the job of a compiler is to generate the XML parameters for each of the hardware primitives. Because the template is intelligent and is able to extrapolate many values from the provided parameters (for example, an interface only needs to be parameterized in one place, then can be cloned repeatedly throughout the design), the set of parameters can be fairly small, though Genesis parameters tend to be very deep (arrays of hashes of hashes, for example). The parameters which specify the behavior of the memory system (programming values inside of a LUT, for example) are only part of the configuration, so the SLAMM compiler must work well with tools to set other memory system parameters, such as a Graphical User Interface (GUI).

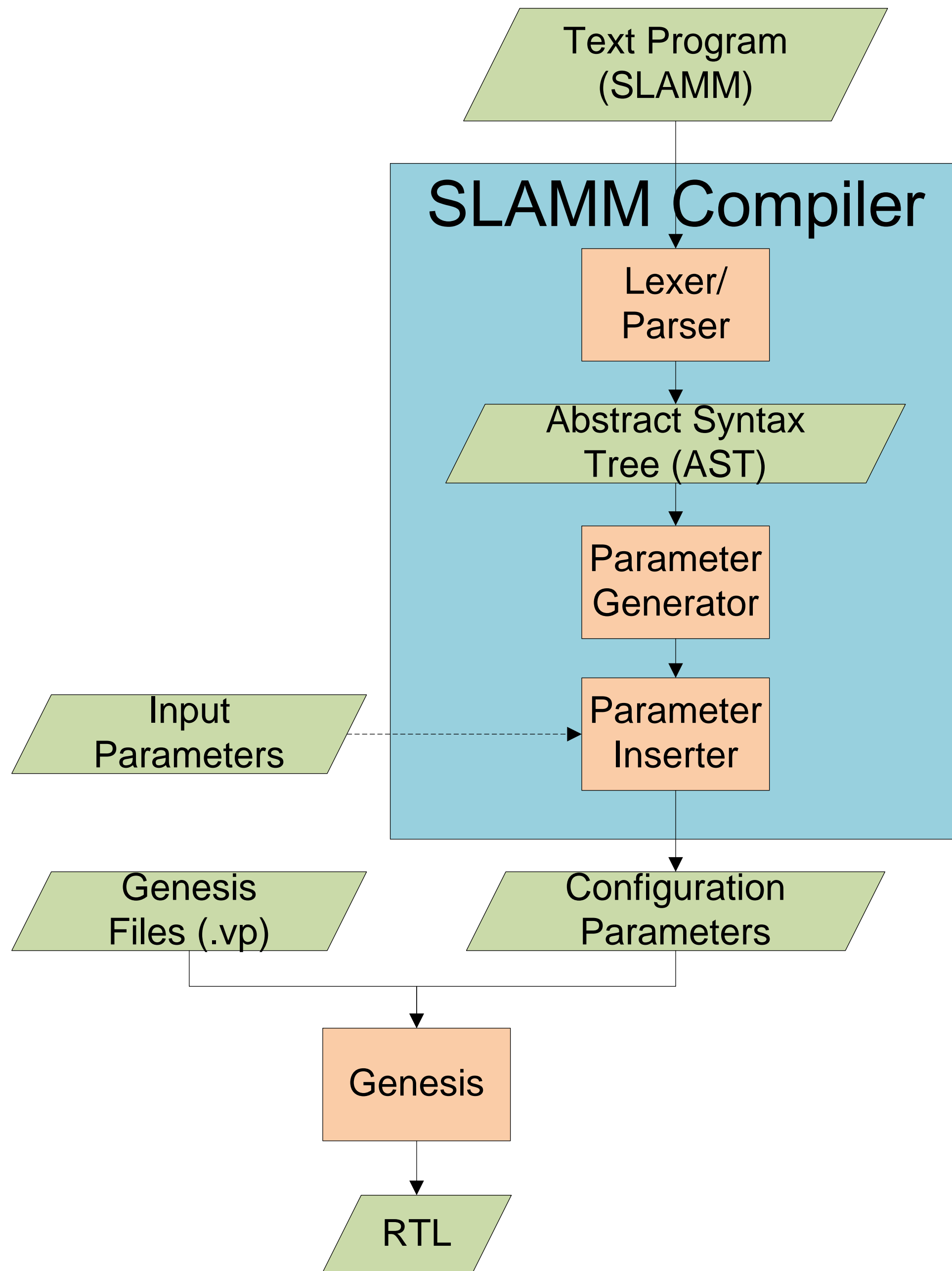


Figure 4.9: SLAMM Compiler Flow

4.2.1 The SLAMM Compiler

Similar to the way the GEMS SLICC project provided a compiler which took a SLICC specification to generate C++ simulator code, the SLAMM compiler can convert the SLAMM specification into parameters expected by the architectural model in order to generate synthesizable hardware. By retaining backwards compatibility with SLICC the simulation and performance modeling capabilities of the language were maintained via the SLICC compiler. SLAMM's compiler flow is shown in Figure 4.9.

The compiler (written in Python) takes as input the SLAMM program, generally as a collection of files. All of the files are lexed and parsed to create an Abstract Syntax Tree (AST) of the design. Then, a series of visitors traverse the tree in order to extract the information relevant to the given architecture. The TypesVisitor extracts information about the different StateMachines in the system, the enumerations (including Events, MessageTypes, Actions, and States), different structures and external types. Next, the InputsVisitor extracts information about what logic is necessary for generating Event triggers (because event triggers are generated by messages arriving on the input ports). This is used to build input interface logic for the Event Trigger TCAMs. Next, the TriggersVisitor generates the actual logic contents for inside the TCAMs, the logic which controls the Event trigger generation. It also handles the logic related to Transitions, in order to configure the TCAMs which determine the next state and actions to take based on current state and event. The ActionsVisitor then generates the logic for each possible action. It determines whether a given action sends a message, and if so, the destination and how to populate the fields of that message.

When the visitors have traversed the AST in order to extract the necessary information, the compiler then generates a number of XML parameters. These are complex data structures which can include a hash of signal information to describe an interface, or all the information needed to generate the inputs, outputs, and programmability of a TCAM. Parameters are output into a set of XML files, but are not quite ready to be input to Genesis. A final step, the `slamm_insert` script, takes the XML parameters and inserts them into the correct place in the existing hierarchy.

The SLAMM compiler can not generate a full XML description from scratch, for several reasons. First, SLAMM restricts its scope to describing the memory hierarchy, and therefore parameters describing, for example, adder widths inside the FPU, are not relevant or controllable through SLAMM. Other tools configure those parameters and the

SLAMM flow allows the parameters to be merged together. Another reason the full XML can not be generated by SLAMM is that SLAMM is agnostic to many aspects of the system, such as Data and Address widths, line size, etc. The SLAMM specification uses abstract Types for signals like Data and Address, and compiles them to parameters with widths like DATA_WIDTH. Only if SLAMM is specifically knowledgeable about a field does it extract a width (enumerations for State, Event, etc. fall into this category).

The SLAMM inserter controls which parts of the system are configured with which SLAMM specification. If there are multiple memory blocks in the system, the inserter puts them into some or all of the blocks, without the SLAMM specification being aware of how many blocks are in the system. If another block is to be specified as a scratchpad memory and not a cache, then it does not get the Cache Coherency protocols inserted. The SLAMM inserter script is a simple Python program which is easy to modify by hand to insert the parameters into the desired modules. If another memory level was introduced, the inserter script would need to be modified to ensure that the new parameters are inserted into the new controllers.

A key role of the compiler in processing the states and message types is to assign values to each, such that the different controllers in the system can agree on the meaning of a certain value. Currently, the SLAMM compiler compiles these values down to raw numbers in the parameters (rather than building something like a Verilog `parameter` out of them). This has a negative impact on the readability of the code, but follows the Genesis model of making sure that all parameters have clearly defined values at compile time. As a debugging aid, therefore, the compiler also adds a parameter to the controller which is a hash of names to values. The hardware template writer can query this information-only parameter to add comments to their code using the actual name of the States, message types, etc.

We've described this high level language and a template to accept it. However, we acknowledge that it may be inefficient or unnecessary to redesign a whole system in its entirety to use this model. The flow we described works well even when some components are not specified in this way, or are only partially specified. To illustrate this point, the next section describes the aspects of our current system template that are controlled by the SLAMM specification, and to what degree.

4.3 Mapping SLAMM Constructs to the Architectural Model

The exact mechanism for the SLAMM constructs mapping onto the architectural model differs depending on the specific part of the architecture. For example, inside the Memory Block the state of a cache line is tracked very differently (in hardware) than in the Protocol Controller. There is no intrinsic reason that this has to be the case, but for historical reasons the system template used a lot of legacy architecture (e.g. a number of small memory mats connected with a connection network) to construct a cache or other memory structure. SLAMM is not concerned with the exact mechanism of state update, as long as the hardware provides some way of describing the CurrentState and accepting the NextState. Therefore, in our current hardware we have to provide mappings back and forth from the CurrentState as stored in the cache line to the enumerations understood and used by SLAMM. In a system designed from the ground up, this would not be the case. To demonstrate this point, in contrast to the more legacy memory blocks, the reply handler and protocol controller modules were architected while considering the input that could be provided from SLAMM and the compiler, so their internal workings are almost entirely described by SLAMM, except for the function of special blocks.

4.3.1 Interfaces Between Blocks

The SLAMM specification generates the information for each interface in the system, as shown in Figure 3.6. It outputs parameters which describe Memory Block to Protocol Controller Request and Replies, Protocol to Memory Block Request and Replies, and Protocol Controller to Network Switch Request and Replies. It also outputs information for Network Switch to Protocol Controller Request and Replies, though those are not currently used because they are for multi-chip systems.

In the current chip generator, the hardware interfaces do not necessarily follow exactly from the SLAMM specification. For example, the crossbar going between the protocol controller and the memory block does not use SystemVerilog interfaces, so it does not naturally accept the interface primitive that the SLAMM compiler knows how to parameterize and that the protocol controller expects to export. However, even the legacy crossbar can make use of the parameters extracted from the interface to pack and unpack the signals going into and out of the crossbar.

Another example of the hardware using the information from the SLAMM compiler in

a less literal way is that the actual hardware width of the Protocol Controller to Memory Block interface should be restricted, whereas the protocol could specify some very large signals (e.g. a cache line). The hardware uses the `message_pipe` primitive to serialize such wide interfaces, transparently to the protocol writer.

4.3.2 Protocol Controller

Figure 4.1 shows what happens inside the protocol controller on reception of a cache miss message, in order to implement the protocol controller's portion of the simple cache protocol in Figure 3.4. The figure shows that inside the protocol controller, almost everything is specified by SLAMM. The fields of the MSHR entry State primitive are described by a special `structure` in the SLAMM program, are extracted by the compiler, and inserted into the protocol controller's XML configuration with the parameter MSHR_ENTRY_SIGNALS. The hardware provides those to the state primitive which it instantiates as the MSHR.

The protocol controller's events and states are provided by name and value to make it easier to agree on these between blocks, since sometimes the State is also a field in a message from another block. In a world fully controlled by SLAMM and the SLAMM compiler this would not be a problem, but the reality is that outside of the core area, other user tools want to refer to States by name rather than an opcode known only to the SLAMM compiler, so providing this mapping helps eliminate errors in these cases.

In order to determine which event has occurred, the Protocol Controller instantiates an Event Trigger TCAM, shown in Figure 3.8. This TCAM is entirely controlled by SLAMM, including determining the inputs to the TCAM. The protocol controller template actually first generates the TCAM, then requests from it the inputs that it needs to determine its logic function. It then makes sure to connect the necessary input signals. The internal configuration of the TCAM is fully specified by SLAMM and inserted hierarchically into the configuration file, so the Protocol Controller does not need to configure it.

In order to determine the Next State and Actions to perform, the Protocol Controller has another TCAM, the Transition TCAM, as shown in Figure 3.9. Its inputs are known: the CurrentState and the TriggeredEvent, therefore the SLAMM compiler does not need to specify the inputs as it does for the Trigger TCAM. Also, the outputs are known to be all the Actions which have already been given to the Protocol Controller, so the Protocol Controller can simply instantiate and connect to the TCAM, and let it handle its own internal configuration.

For each output interface in the Protocol Controller, a translation block is instantiated, shown in Figure 3.11. The SLAMM compiler is responsible for generating both the input interface to each translation block, and the internal logic. This also means that for any structures inside the translation block (LUTs, TCAMs, TCAM Muxes), the SLAMM compiler has to insert parameters for them inside the XML. The input interfaces are extracted by determining every possible driver of the fields of the output messages, by examining all the enqueue actions in the protocol, and making sure that all necessary inputs are provided. Also input are the actions which actually perform an enqueue for this output interface. One thing to note is that one source of data for output messages can be from performing actions on message blocks. For example, calling `getData(Address)` on the LineBuffer object. One thing notable is that SLAMM knows how to handle, for State instances, the `get*(*)` method. In the above example, the compiler will know to do a lookup based on the Address field of the State, and connect the Data field to the input to the Output Translation TCAM (see Figure 3.15).

4.3.3 Memory Block

The Memory Block uses a legacy way of storing State and Data (in memory mats) and has a highly parameterized design outside of the scope of SLAMM for specifying things like size of cache or number of ways of associativity. However, in terms of receiving instructions from the Protocol controller, determining what actions to take, and responding appropriately to protocol controller requests, it is fully controlled by SLAMM. Its core structure is essentially the same as that of the Protocol Controller, with the same TCAMs and Translations. However, because the memory has a cycle latency to access its state (it must be read out of a memory macro), different buffering and delay structures are included in the hardware template for the memory block.

The Memory Block also has some issues where it needs to interface between the SLAMM-controlled world and the hand (or other tool) generated code. Here the human-readable mappings from State and Events come into play as users create the mappings from protocol controller opcode to low-level memory opcode. The SLAMM compiler does provide a mapping from protocol controller operation to internal cache block operation (an identity mapping, as the known protocol controller operations have been hand coded into the cache template's defaults).

Inside the Memory Block, the cache itself is currently handled as a highly specialized

magic block, as it knows how to select an eviction candidate, defines aliases for getData, getLineState, getEvicitionState, etc.

4.3.4 Reply Handler

Because the reply handler enforces that the transactions it tracks complete in order, it uses the state FIFO primitive to track outstanding operations. It also uses a TCAM to describe the messages it sends to the Protocol Controller. A separate unit which stalls the issuing processors if there is a dependency in outstanding operations in the Reply Handler uses the TCAM cloning capability to track the state in a similar fashion in order to determine if there is an outstanding load which blocks on the current action. Otherwise, the reply handler is configure with the GUI tool.

4.3.5 Off-Chip Memory

The off-chip memory which the protocol controller communicates with is currently hand-coded. Structures inside the test bench translate the SLAMM-specified interfaces into the interface used by the low level off-chip memory. In theory these are parameterized and could be more flexible, there is just not currently interest in configuring these modules.

4.4 Extending a Protocol with SLAMM

In our simple example in Figure 4.1, we only described what would happen on a read miss, though we can define a message type for an upgrade (write) miss. In order to specify the path for a write miss, the user writes the cases in isolation, without regard to the previous read-miss description, then merge the two files. That is, they should not conflict, except for taking into account what happens if an upgrade miss comes in while a read miss is being handled; in general it is easy for the SLAMM hardware to NACK or block if such complications are not desirable. More states would be added as well as new sources to the messages, but SLAMM will handle all this automatically.

What about a more interesting extension of the protocol, where we change the underlying protocol somewhat? In Section 2 we described a cache, PLCache [47], which aimed to add a "Lock" bit to the protocol in order to prevent cache timing attacks. Lines which were locked could not be evicted (if all lines in a way were locked, then no evictions or refills could be performed on that line). New processor opcodes are added to LOCK and UNLOCK cache

lines, and lines with the Lock bit set should be considered in new states of LOCKED or LOCKED_MODIFIED. In addition, lines that are locked know who their Owner is because it is recorded in their state (the Owner of the line is allowed to evict it regardless of its locked state).

In order to implement this addition with SLAMM, first the user would add the new states LOCKED and LOCKED_MODIFIED. They would consider what events would cause transitions to and from these new states, and update the transitions in their specification accordingly. To the MSHR entry fields which get mapped to State structures, they should add the Owner field. If the SLAMM compiler observes the State is being examined based on the Owner field in addition to the Address field, it would do a lookup based on both. Finally, the user would need to modify the internal eviction rules to consider the Locked bit. This is not currently described in SLAMM, so the findEvictionCandidate function's interface is not really changed, except that it might fail entirely. Therefore, another signal or reply type can be added which indicates that no eviction candidate was found (or that the eviction candidate state was Locked), in which case the Protocol Controller would have to return the data without refilling the line.

Although this seems like a lot of changes, consider what the user would have to do if they did not have SLAMM. Simply adding a field to one interface in SLAMM (changing one line of code) changes or adds over 450 lines of hardware Verilog code in 29 different modules (over 10% of the modules in the system level design). Adding a condition based on that interface (3 more lines of code if you like a lot of white space in your `if-then-else` statements) changes over 60 additional lines of Verilog in 4 modules. A user would have to make these changes by hand and handle them all correctly, a process which from experience is painful, time consuming, and a matter of guesswork.

Even a protocol specified in SLAMM could contain errors at the Protocol Specification Level (Figure 2.11). These errors could stem from programming errors in the specification, or bugs in the underlying hardware template or compiler flow. Therefore, the next section describes a tool to aid the verification of the output of the generator.

Chapter 5

Relaxed Scoreboard

The previous sections have described a flexible hardware architecture which can be programmed to implement different memory protocols by reprogramming basic primitive blocks within a variety of controllers. Although the conversion from high level specification to parameters/microcode is automated, there are still plenty of opportunities to introduce errors into the implementation. A programmer writing in SLAMM code could misunderstand something about the way the architecture behaves, or make an error when writing their protocol. In order to verify the protocol, the user can write a simulator which also compiles the SLAMM code in order to run it (without a full hardware implementation). This is exactly what was done by SLICC in order to run it through the GEMS simulator. However, even if the hardware and simulator match, the user may have an error in the protocol. There are many tools and techniques for formally verifying memory protocols at a variety of levels, but tools for verifying their implementations (in hardware or software) are also essential. In this chapter we describe one such tool for verifying memory system implementations at a high level, the Relaxed Scoreboard[1].

Most practical approaches to validation of complex systems have focused on a framework based on either race-free diagnostics or pseudo random test suites combined with a golden model (also known as a scoreboard) of the memory system behavior. While this methodology is widely used and works well, constructing the scoreboard for a modern system is a difficult task. One of the factors that makes it complex is that memory consistency models such as sequential consistency (SC) specify rules and axioms that are easier to describe by a non-deterministic state machine—they do not completely specify the memory system's

[1] The work presented in this section was done in conjunction with Ofer Shacham.

behavior. As a result, a given consistency model may have multiple correct implementations that perform differently, and thus need different golden models. This coupling causes two problems. First, one needs to create a new golden model for each implementation, and second, it is hard to keep the validation model completely separate from the implementation, which can lead to correlated errors. This problem is only made more challenging for a flexible hardware architecture such as the one proposed here, because a new reference model would need to be constructed for every new configuration.

Recent attempts, such as TSOtool [19], take a different approach to the problem of memory verification. Instead of dealing with the complexity of the implementation, TSOtool does a post-mortem analysis of the processors' traces. This approach checks that the observed trace values are logically correct with respect to the consistency model. Since it does not specify what the output should be at each cycle, or even what the ordering must be, it reduces the coupling between the verification model and the design details. The key insight is that the undesirable verification-design coupling can be broken by creating a checker that allows multiple output traces to be correct. We leveraged this insight to create a new approach toward validating memory system behavior, the Relaxed Scoreboard.

The Relaxed Scoreboard is a verification methodology that attempts to come as close as possible to verifying the temporal behavior of a CMP memory system implementation, while avoiding exponential complexity. Like a traditional scoreboard, the relaxed scoreboard is constructed to be an intuitive and simplified temporal model of the memory system, but like TSOtool, it is not tied to a specific implementation. The decoupling of the relaxed scoreboard from the implementation is achieved by having a set of multiple possible values associated with each memory location, similar to earlier work by Saha et al [33] and TSOtool. This set of values is constrained by rules specific to the implemented memory protocol, and includes all values that could possibly be legally read from this address by any processor in the system.

We find that by using this relaxation (keeping a set of possibly correct answers), a relaxed scoreboard methodology introduces a number of traits that are important for efficient RTL design and verification. The construction of the scoreboard is derived directly from the relevant consistency model properties. Each of those properties can be considered separately, allowing a protocol designer to re-use rules that apply to their protocol, remove those that do not, and provide a clean interface for writing new rules.

This enables the verification environment to be developed incrementally along with the

design, rather than requiring a complete model on day one. In contrast to static post-mortem trace analysis algorithms, the relaxed scoreboard is designed to be a dynamic on-the-fly checker, meaning that any error will be reported immediately, saving valuable human and compute time. Furthermore, the combination of dynamic runtime analysis plus the scoreboard's incremental construction enables the user, at later design phases, to incorporate key information from the design (such as exact arbitration time) into the scoreboard. While at the beginning of the design/verification cycle the number of acceptable results might be large, this set can become smaller as more sophisticated checks are added to the scoreboard, and can, if needed, turn into a tight, accurate model.

The next section formally describes two different memory consistency problems: verifying sequential consistency both with and without temporal information about the memory operations. Section 5.2 then describes the basic approach for creating a relaxed scoreboard, and demonstrates how it can be used to address the consistency problems described. Section 5.3 extends the discussion to a real design example, where relaxed scoreboards were used to verify the implementation of an actual CMP design, the Stanford Smart Memories Project, for both Relaxed Consistency and Transactional Coherency and Consistency (TCC). The section also describes how modular rules were used for the different memory protocols in Smart Memories and could be extended for new memory protocols.

5.1 Problem Definition

Verification of a shared memory system is in essence the attempt to prove that the hardware complies with the mathematical definition of the coherence and consistency model from a programmer standpoint. For example, deciding whether a set of processor execution traces complies with sequential consistency is known as *Verifying Sequential Consistency* (VSC) [13]. Similar definitions apply for other consistency models [8]. When dealing with RTL/architectural verification, as opposed to post silicon verification, an attractive verification approach is to leverage not only the values observed on the system's ports, but also their temporal information—the time at which they were observed. By using temporal information, a checker can also flag errors that obey the consistency model but should not occur in real hardware. The temporal version of VSC is known as *Verifying Linearizability* (VL) or *Atomic Consistency* [14].

The following are the formal definitions of Verifying Sequential Consistency and Verifying Linearizability, as described by Gibbons and Korach [14]. In layman's terms, each definition describes a number of sequences of reads and writes to a set of addresses, and the question is whether those sequences are legal given the memory consistency model.

Verifying Sequential Consistency is based on respecting program orders and the read/write semantics, while ignoring the time at which transactions occur.

> GIVEN: A set of addresses' A, a set of data values D, a finite collection of sequences $S1, ..., Sp$, each consisting of a finite set of memory operations of the form "$read(a, d)$" or "$write(a, d)$", where $a \in A$ and $d \in D$.
>
> QUESTION: Is there a sequence S, an interleaving of $S1, ..., Sp$ such that for each $read(a, d)$ in S there is a preceding $write(a, d)$ in S with no other $write(a, d')$ between the two?

Informally, if we write each read/write operation on an index card, then make an ordered stack for what each processor sees, sequential consistency requires that we should be able to interleave the stacks from each processor, in some way, and still get a global ordering that makes sense.

Verifying Linearizability adds the further constraint that the schedule S must respect the time intervals for the operations.

> GIVEN: A set of addresses A, a set of data values D, a set of finite collection of sequences $S1, ..., Sp$, each consisting of a finite set of memory operations of the form "$read(a, d, t_1, t_2)$" or "$write(a, d, t_1, t_2)$", where $a \in A$ and $d \in D$, and t_1 and t_2 are positive rationals, $t_1 < t_2$, defining an interval of time such that all intervals in an individual sequence are pairwise disjoint, and t_1 and t_2 are unique rationals in the overall instance.
>
> QUESTION: Is there an assignment of a distinct time to each operation such that 1) each time is within the interval associated with the operation; 2) for each $read(a, d, \tau_1, \tau_2)$, there is a $write(a, d, t_1, t_2)$ assigned an earlier time, with no other $write(a, d', t'_1, t'_2)$ assigned a time between the two?

Informally, now the "index cards" for the protocol have start and end times written on them. The interleaved stack must not only make sense, but it must be possible to spread the cards out on a timeline so that each card lies on a point between its start and end time.

Similar definitions can be applied to other consistency models such as Total Store Order, Weak Ordering, etc. One should note that in practice, most CMP architectures would allow non-blocking writes, and even multiple outstanding reads. This means that a black-box verification environment will not have the end-of-interval time stamp for *write* operations, and that all operations may not be pairwise disjoint. The practical implication is that a checker may need to keep track of more than P concurrent accesses, where P is the number of processors in the system. One such architecture is Stanford Smart Memories [40, 12], on which the relaxed scoreboard was evaluated.

Gibbons and Korach proved that VSC is NP-Complete [14]. Cantin, Lipasti and Smith extended the proof to all other commonly used consistency models [8]. Moreover, the problem is NPC even when any of address-range/number-of-processors/accesses-per-processor factors is bounded. Although VL was also shown to be an NPC problem, the limiting factor for VL is the number of processors that are accessing a shared address, rather than the length of the trace.

In our goal of verifying a memory system, we want to have the tightest bounds on correctness possible, in order to catch as many errors as possible. We argue that for the purpose of hardware verification, VL is a stronger correctness criteria since any set of traces that would violate a checker for VSC will also violate a checker for VL. Limiting verification requirements to VSC only makes sense if there is no time information provided (for example, in a highly distributed system where there is no reliable global timestamping). Even if there are no time *limits* on an operation, with the timestamps assumed in VL, there is still the information provided by the fact that time can not run backwards, and things that are known to happen later in time can not influence things that happened in the distant past. Because of this, there can be traces with errors that violate VL but do not violate VSC. The following set of temporal sequences demonstrates such a scenario, which encourages us to use VL if we have time information available to us.

Time	S1	S2	S3
10	A:RD(a,5,10,11)	D:WR(a,1,10,11)	G:WR(a,2,10,11)
20	B:RD(a,5,20,21)	E:WR(a,3,20,21)	H:WR(a,4,20,21)
30	C:RD(a,5,30,31)	F:WR(a,5,30,31)	I:WR(a,6,30,31)

* Assume MEM[a]=5 at time 0

** This example illustrates a 'stuck-at' error case;
despite multiple competing writes from other processors,
S1 only ever reads the value of '5' from address 'a'.

In the counterexample above one can easily find a global ordering that complies with SC (E.g.,{G,H,I,D,E,F,A,B,C}) . However, no ordering can be found that complies with VL. This implies that a checker with temporal information will find more error cases than a post-mortem checker without temporal information. The latter might not find errors that violate causality as in this example. Section 5.2.1 will show another example of a property violation (write atomicity) that can only be found by a checker that leverages temporal information. Therefore, going forward we will assume we have temporal information when creating our checker.

5.2 Relaxed Scoreboard

To verify the linearizability of a CMP implementation, the Relaxed Scoreboard is built as a global online design checker. As the design is running other tests (for example, application code), the scoreboard monitors all transactions going into the system, and all the transaction output from the system. It not only checks whether these outputs are allowable, but it uses the outputs to modify its internal tracking of what future results are allowable. As noted before, the relaxed scoreboard does not compare an observed value to a single known correct result. Instead, it keeps a set of possible results and ensures that the observed value is within this bounded set. It is an oracle that can answer the following question: "Does the value *Val*, that was observed at interface *ifc* of the design, exist in the group of allowed values for that interface, at that given time?" In this sense, the relaxed scoreboard is simply a data structure of groups, which can answer queries and receive updates.

In order to understand why a single value might be hard to predict but a bounded set is not, one can think about a simple 4-input round-robin arbiter. In order to predict who would win the arbitration, a golden model would have to "know" or imitate the internal state of the arbiter. In contrast, in order to determine the set of possible winners, a relaxed scoreboard only needs to know which inputs are asserting a request and which inputs have already been granted (in previous rounds). It would then perform simple accounting operations to keep the set of possible winners as tight as possible. Note that in addition to simply checking the DUT outputs, the scoreboard uses those outputs to reduce the allowable future outputs.

Verification of a memory system in a shared memory multiprocessor is particularly challenging because of the inherently non-deterministic and timing-dependent execution of memory accesses, which inevitably produces race conditions. The sequence of memory

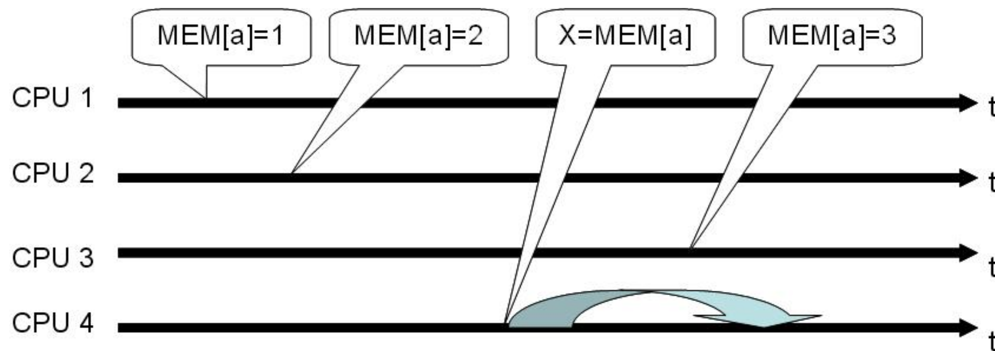


Figure 5.1: Non-determinism of a CMP Memory System.

accesses from each processor might depend on how a prior race condition was resolved. Figure 5.1 shows a simple example of a race condition in a four-processor system. Processors 1, 2 and 3 are initiating writes to address a while processor 4 is reading the value saved at that address. The data is returned some time after the load started, as noted by the curved arrow. Depending on the exact arbitration mechanism, each of the values 1, 2 or 3 might be returned as the result of the load instruction.

In order for the relaxed scoreboard to be an efficient verification tool, we define two requirements:

1. Bounded Uncertainty: The set of possibly correct answers must be kept bounded. If this set grows in time, then the testbench as a whole loses efficiency. This is the most important requirement of the relaxed scoreboard.

2. On-The-Fly: Errors should be detected on-the-fly, as close as possible (in time) to their origin. Since the relaxed scoreboard is designed with big, complex designs in mind, it must recognize the need of both designers and verification engineers for a fast turnaround time.

Most directed tests use an approach in which there is only one known allowed result, by using locks, long delays, or other mechanisms to remove the ambiguity in a test. On a test of this nature, a relaxed scoreboard should be able to perform as well as a deterministic checker, in order to be considered a useful tool. This means that the set of allowable values should reduce to 1 for a test without ambiguity, once a sufficient number of checks and updates are added to the scoreboard. Our results with this technique (discussed in Section 5.3) show that in directed tests, the relaxed scoreboard does have a small set of

allowed values, almost always 1. But, even in “interesting” test cases where there is a lot of collisions on addresses, the set of allowed responses stays small and bounded. The reason this is possible is the addition of updates to the scoreboard : using the observed outputs in addition to the inputs in order to limit the size of the allowed set.

To meet the above requirements efficiently, the relaxed scoreboard implementation uses an internal, easily searchable data structure. The main purpose of the scoreboard’s internal data structure is to keep a set of possibly correct values, sorted by address and chronological time. Each entry in the scoreboard’s data structure is associated with a sender ID, value, and start and end time stamps, as observed during runtime on the relevant processor interface. In addition, each entry contains a set of expiration times, one for each possible interface on which this transaction may be observed. Upon arrival of a new transaction from a monitor, the scoreboard performs a series of *checks* followed by a series of *updates*. The checks produce the scoreboard’s oracle answer, based on the values stored in the data structure, of whether that operation is possible. The updates use the DUT output to update the internal data structure with the new information, reducing the set of answers that the scoreboard considers as correct for future operations. Updates also reduce the uncertainty in the range of time that an operation completed.

Updates reduce the set of possible values held in the data structure, but the scoreboard can perform simple checks even before any updates are added. For example, the most useful check is a check of causality (a value that is read from an address must have been previously written to that address). In this very loose scoreboard, any stored value would be considered as correct. Under such a simple scoreboard, the check would quickly become ineffective and the data structure would explode in size. Thus, updates are the completing and crucial part of the scoreboard. Updates use the rules of the implemented protocol to reduce the set of possible values.

Updates and checks are independent and modular, so for a new or different protocol, checks and updates can be swapped in and out, and new ones can be written. The new checks and updates can be added to verify different aspects of the specification, or existing checks and updates can be made more effective by considering more details of the implemented system. This characteristic allows the verification effort to concentrate on the simplest and easiest to detect errors first, and gradually move towards more sophisticated design problems. This implies that while at the beginning of the design and verification cycle the number of acceptable results might be large, this set later becomes smaller, evolving

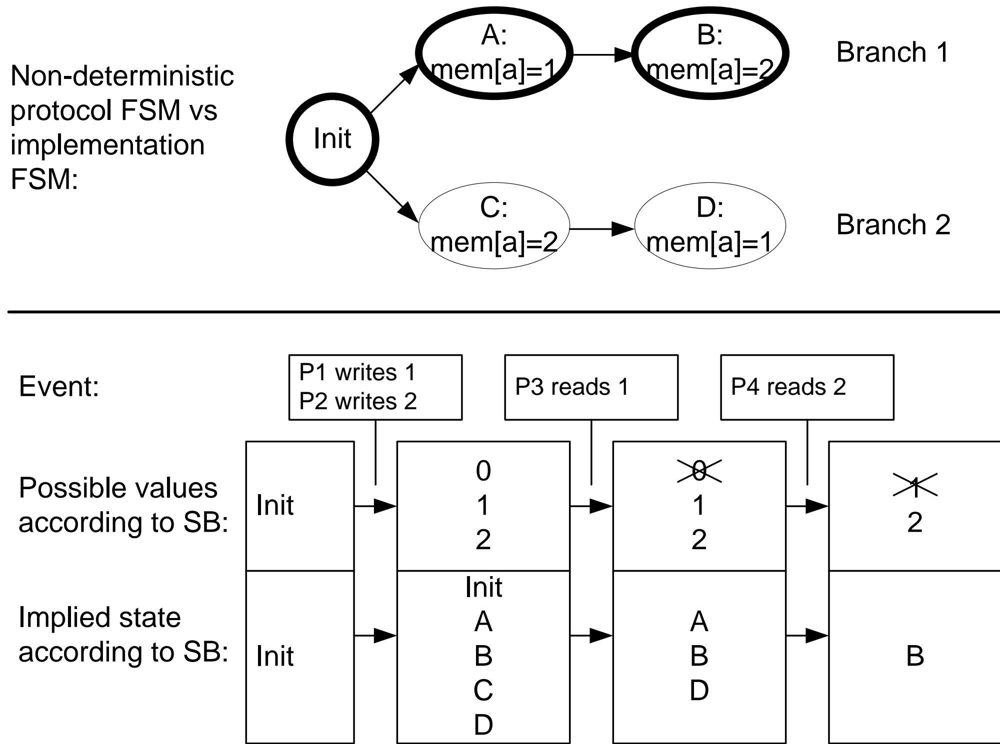


Figure 5.2: Write atomicity example.

towards a tight, accurate model. Overall, the relaxed scoreboard is essentially a set of global protocol-level assertions. The assertions are constructed to verify that certain rules of the protocol are followed at a high level, without actually relying on or examining the implementation details.

Sections 5.2.1 - 5.2.3 show a few example properties that a user trying to verify their protocol's implementation may select. These demonstrate the use of the relaxed scoreboard in verifying implementations of shared-memory protocols. In the examples, *write(a,d)* and *read(a,d)* are used respectively to denote a write or a read of data d from address a. To make the consistency problems easier to see in these examples, reads are assumed to complete in less than 10 cycles, and *nop*s indicate operations that are not relevant for the property in question.

5.2.1 Write Atomicity

As a run-time checker, the scoreboard needs to determine whether a given operation corresponds to a legal transition in the machine state. As our first example, let us examine how a relaxed scoreboard would check for write atomicity, which is part of many consistency models [2]. In order to detect write atomicity violations, we convert the protocol property into a check and update, and add them to the relaxed scoreboard:

> PROTOCOL PROPERTY: All processors see writes to the same location in the same order: that is, each write constitutes a single serialization point. If there is a store that is observed by one processor, it should be observed as completed by all processors.
>
> UPDATE: When one processor loads a value from a certain location, mark all previously committed stores to this location as invalid after the load completion time.
>
> CHECK: A load can only return a value which was valid at some point between its start-of-interval and end-of-interval.

The following is an observable write atomicity violation. P5 incorrectly reads a value of a=1 *after* P4 has verified that the a=2 update definitively occurred by time t=30.

Time	P1	P2	P3	P4	P5
10	WR(a,1)	WR(a,2)	nop	nop	nop
20	nop	nop	RD(a,1)	nop	nop
30	nop	nop	nop	RD(a,2)	nop
40	nop	nop	nop	nop	RD(a,1)

* Assume MEM[a]=0 at time 0

** Assume read operations complete in less than 10 cycles

In this example, Processors 3, 4, and 5 disagree on the ordering of the store to address a. Figure 5.2 illustrates the operation of the relaxed scoreboard for the above code sequence. The top part shows portions of the specification's non-deterministic state machine that correspond to the code above. It shows that one of two sequences can exist: either P1 wrote 1 and then P2 wrote 2 or vice versa—write atomicity is maintained in both cases. The portion of the figure that is drawn in bold illustrates the corresponding implementation's state machine (which is deterministic). The bottom part of the figure shows the state of the relaxed scoreboard with respect to the same code: The scoreboard identifies two writes and marks both values as possible answers. This corresponds to the state machine being in either state $Init$, A, B, C or D. When the first read is reported to the scoreboard,

the scoreboard deduces that the design can be in either state A, B or D, and when the second read is reported, the uncertainty window collapses to a single allowed value. In this example, the scoreboard will immediately identify the read by processor 5 as an error since the returned value is no longer in its list of allowed values for that address. It is important to note that a trace analysis checker that does not use temporal information will not identify this set of traces as erroneous. Without the time information, the read by P5 could be assumed to have taken place before the read by P4.

5.2.2 Transaction Isolation in TCC Memory Model

To provide another example of how to convert protocol rules into checks and updates we examine transactional memory. *Transactional coherence and consistency* (TCC) is a memory model that has been proposed to simplify the development of parallel software [18]. In TCC, the programmer annotates the code with start/end transaction commands, and the hardware executes all instructions between these commands as a single atomic operation. If a transaction conflict is detected, such as one transaction updating a memory word read by another, the latter transaction is considered as violating and must be re-executed.

In order to track the state of a TCC system, the relaxed scoreboard must be able to determine when transactions begin, commit or abort. Fortunately, as part of the software-hardware contract, the timing of these events is determined by a TCC runtime system using special marker instructions to signal the state of a particular processor and transaction. The relaxed scoreboard can use this information for its own operation. The scoreboard's data structure also needs the ability to keep track of multiple transactions simultaneously; all written values must be kept and associated with their initiating processor until the time of commit. Similar to the example discussed in Section 5.2.1, the scoreboard does not need to know the exact timing of the events in order to check for end-to-end properties. For example, the scoreboard does not know when a store becomes visible to other processors; it assumes that it happens sometime during commit.

The *Transaction Isolation* property means that when a processor is executing a transaction, the transaction's intermediate state is hidden from other processors in the system. They can only observe the transaction's state before a transaction start or after a transaction commit. As in the previous example, in order for the relaxed scoreboard to check for transaction isolation, we convert the protocol property into an update:

PROTOCOL PROPERTY: A transactional store cannot be observed by other processors

unless the transaction commits successfully.

UPDATE: When a store is issued during a TCC transaction, leave it as invalid for all but the issuing processor. Upon observing transaction commit, validate it for other processors as of commit time.

CHECK: A load can only return a value which was valid at some point between its start-of-interval and end-of-interval

The following sequence is an example of a violation of transaction isolation property:

Time	P1	P2
10	START_TRANSACTION	START_TRANSACTION
20	WR(a,1)	nop
30	nop	RD(a,1)
40	START_COMMIT	

* Assume MEM[a]=0 at time 0
** Assume read operations complete in less than 10 cycles

In this example, the scoreboard would not allow the value 1 for P2's load, as it has not observed the commit event from P1. Thus on P2's load it would raise an error.

5.2.3 Store Ordering in a TSO Memory System

A design that implements Total Store Ordering (TSO) ensures that all stores from a processor complete in program order, as seen by all processors (this is different from weak consistency, which makes no guarantees about the ordering of stores to different addresses). To facilitate verification of store ordering, we add another update to the scoreboard:

PROTOCOL PROPERTY: All stores by a processor must commit in issuing order, as seen by all processors

UPDATE: When a store by processor Pi is observed by a processor's load, all previous stores by Pi to all addresses should be marked as committed. If more than one store by Pi exists for a given address, mark the latest as committed, and invalidate older ones.

To analyze the scoreboard actions when used for TSO, consider the following example of a TSO violation:

Figure 5.3 illustrates the operation of the relaxed scoreboard for the above code sequence. The top part shows the change in state of the machine given the TSO memory model. The

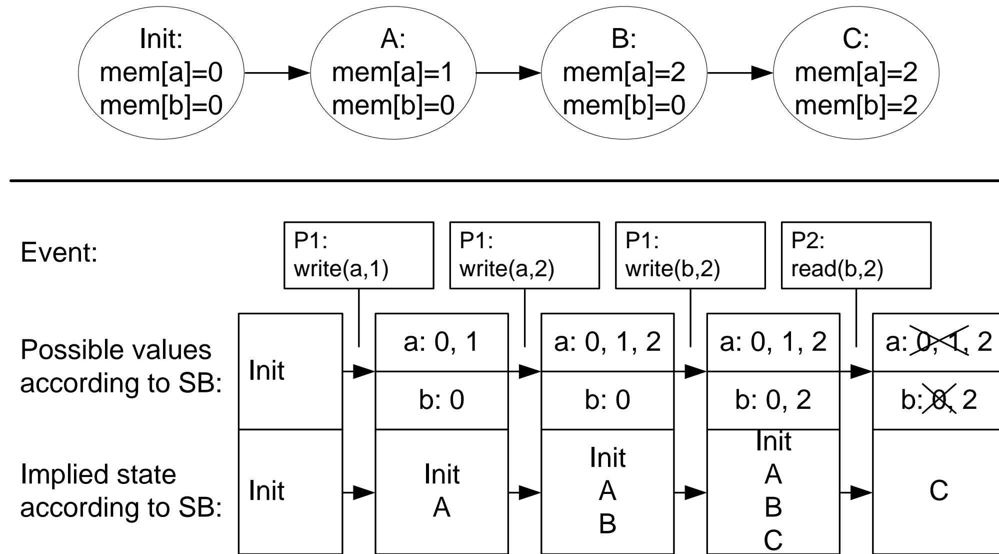


Figure 5.3: Total Store Order Example.

Time	P1	P2
10	WR(a,1)	nop
20	WR(a,2)	nop
30	WR(b,2)	nop
40	nop	RD(b,2)
50	nop	RD(a,1)

* Assume MEM[a]=MEM[b]=0 at time 0
** Assume read operations complete in less than 10 cycles

relaxed scoreboard cannot deduce the exact state, but it can deduce a set of possibly correct values, as illustrated in the bottom part of the figure. By observing the three writes one can deduce that the state is $Init$, A, B or C. When the first read is reported to the scoreboard, the design can only be in state C, therefore the uncertainty window collapses to a single allowed value. Finally, as the second read by P2 is observed, the scoreboard will immediately indicate an error since the returned value is no longer in its allowed list.

The three previous examples showed how it is possible to write simple updates and checks for the scoreboard to verify different aspects of memory implementations. Appendix A contains more implementation details, and for a more detailed analysis of the algorithmic complexity and completeness of the Relaxed Scoreboard, the reader is referred to [37].

One should also note that the relaxed scoreboard can be used to verify a subset of the consistency model properties (thus obviously incomplete). Our experience shows that even in those cases where not all axioms are translated into checks and updates, it was useful in finding elusive bugs that could not have been found using race-free or self-checking test vectors. The next section describes the results of using the relaxed scoreboard to verify several memory models.

5.3 Evaluation

We applied the relaxed scoreboard to the verification of the Stanford Smart Memories chip multiprocessor architecture and evaluated its effectiveness based on the number and complexity of the design errors that it could reveal, in addition to the impact that it had on the overall verification/simulation environment.

5.3.1 Scoreboard Design for Smart Memories

A relaxed scoreboard was used to help verify the design of the Smart Memories chip described in Section 2.2. When configured as a shared memory CMP, the Smart Memories system implements a hierarchical version of the MESI protocol. The shared memory consistency model [2] is a variation of Weak Ordering (WO), which maintains write atomicity. Processors are allowed to issue non-blocking writes to memory, which can be overlapped and completed out of program order. On the other hand, reads are treated as blocking operations and stall the processor until the data is returned. When configured as a transactional memory CMP, the Smart Memories system implements the TCC protocol [18], briefly described in Section 5.2.2.

A relaxed scoreboard was used to aid in the verification of Smart Memories. It was implemented as an object oriented programming class using OpenVera. Vera's Aspect Oriented Programming (AOP) capability was leveraged to connect the scoreboard into the existing verification environment, which already included code to monitor key interfaces. For cache coherency, the scoreboard's data structure contained an associative array of queues, one queue per writable address in the system. For TCC, the scoreboard also maintained a queue for each processor, containing pointers to the transaction's read and write sets, and flags to indicate the state of the current transaction. Checks and updates, a superset of those shown in Sections 5.2.1 and 5.2.2, were written based on the protocol rules

of [2] and [18] respectively.

Due to the scoreboard's black-box nature, it was usable without modification across 25 different cache-coherent configurations and 15 different TCC configurations. Moreover, it was applied to both single-quad (8 processors) and four-quad (32 processors) testing environments, where the only difference was the number of monitor instances that fed the scoreboard with information.

In order to demonstrate the flexibility of the system, we configured the scoreboard to check for (depending on the hardware's configuration), Cache Coherency, Transactional Coherency and Consistency, and Total Store Ordering. For example, adding Total Store Ordering checking to the Cache Coherency checks and updates required writing a single new update for the scoreboard, only 100 lines of Vera code. While this initial version was correct, the data structure was rather inefficient, so we also introduced a second lightweight data structure to track outstanding stores and a short update to populate it.

5.3.2 Quality of Results

To demonstrate error cases that can be revealed by using a relaxed scoreboard, Table 5.1 summarizes classes of errors that the relaxed scoreboard found in the Stanford Smart Memories design. One should note that the errors described in Table 5.1 were found *after* multiple runs of self checking diagnostic tests were used to identify "simple" errors.

Given the complexity of the errors described in Table 5.1, and the fact that some of the errors would have evaded previous verification methodologies, we found the relaxed scoreboard to be a useful debugging tool and another weapon in the arsenal against memory system errors. Using the scoreboard simplified test generation, as tests no longer had to be entirely self-checking. This allowed the test suite to include truly random tests, where before it was limited to tests with only false sharing or strictly synchronized accesses between processors. The random tests revealed subtle bugs, such as erroneous ordering of back-to-back stores to the same address. Moreover, the efficiency of the self-checking tests increased because the relaxed scoreboard was able to detect intermediate errors, even when the final output was correct.

For Cache Coherence we were able to detect many problems related to memory ordering and corruption. These errors existed in many parts of the design: the Cache Controller, Load Store Unit, Memory Controller, and in configuration of our programmable hardware. In the TCC mode, the scoreboard was able to detect errors such as faulty commits and

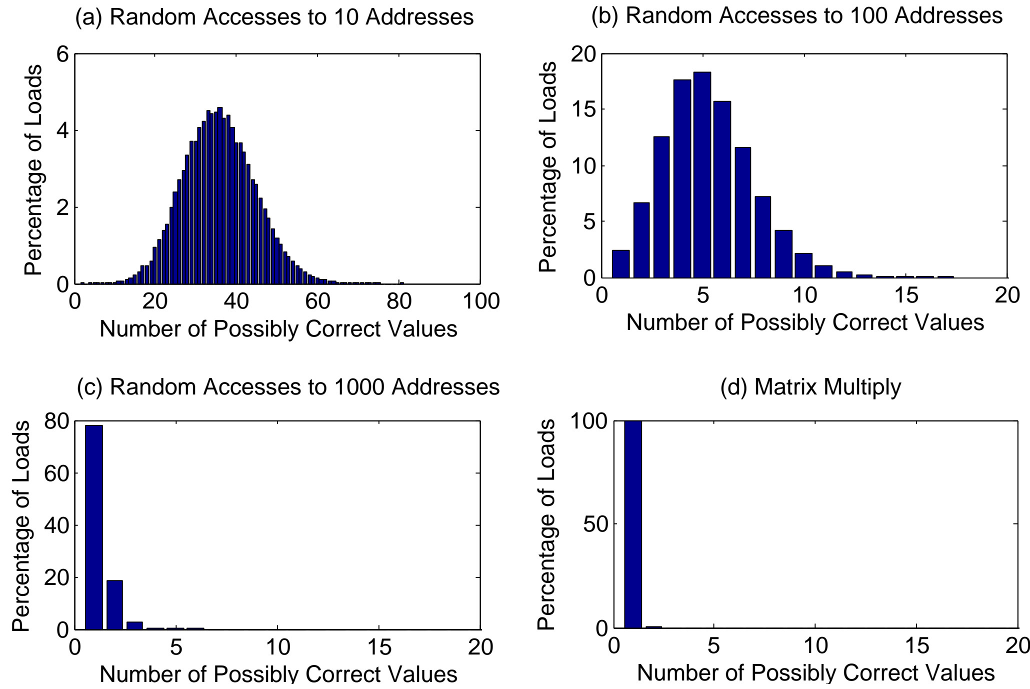


Figure 5.4: Histograms of uncertainty for applications running on 32 processors.

violations. These included a very subtle case where a transaction violated unnecessarily due to an error in the manipulation of state bits. Since this was a performance error, it would have otherwise remained undetected by any self checking test. Another type of error that the scoreboard detected in TCC mode was runtime sequencing problems (software library errors). These included cases in which two start transaction markers were observed without an end/violate transaction marker in between.

The same basic approach served to validate very different memory system models, as well as different implementations of each model. This generality was enabled by the flexible and incremental addition of checks and updates. Table 5.2 lists the checks and updates currently available, while Appendix A provides more information.

5.3.3 Performance and Overheads

A useful measure of the effectiveness of the relaxed scoreboard is the size of the set of possibly correct values (referred to as the uncertainty window). Figure 5.4 shows the size of the uncertainty window for several test-runs on 32 processors. Each subfigure shows

Table 5.1: Classes of errors found by the relaxed scoreboard (LSU=Load-Store Unit; PC=Protocol Controller; MC=Memory Controller; Net=Network)

Error Description	Error Location
Cache Coherency and Consistency	
Program order violation: processor writes new value, then reads the old value from the same address	LSU
Program order violation: later store value is overwritten by older store from the same processor	LSU
Soft processor stall doesn't work	LSU
Back-to-back loads from the same processor return different values	LSU
Coherence violation: load from a different processor returns zero instead of valid data	PC
Instruction fetch returns incorrect value	LSU
Instruction fetch returns incorrect value	Boot sequence
Load returns a no-longer valid data	MC
Load returns incorrect data	LSU
Load returns incorrect data	MC
Load returns incorrect data	PC
Load returns incorrect data	Compiler
load returns X during boot/setup	Boot sequence
Synchronized load returns incorrect data	MC
Synchronized load returns incorrect data	PC and PC config
Transactional Coherency and Consistency	
START_TRANSACTION command called twice	TCC runtime
Missing COMMIT command	TCC runtime
Unnecessary TCC violation	PC
TCC coherent load returns stale data	Net Config
TCC committed value is lost	Net Config
TCC transaction is not violated, missed dependence	PC
TCC coherent load returns wrong data	PC

Table 5.2: Checks and Updates for Memory Protocols, as implemented in the Relaxed Scoreboard for the Smart Memories project.

Name	Description
	Checks
Check Last Received	For loads, check that the last observed value exists in the set of allowed values. Special cases for the first observed access to an address.
Check Count Valid	A Bookkeeping check which tracks the size of the uncertainty window each time the checks are run.
	Updates
Update Simple	When a processor N stores a value to address A, no earlier stores to A are valid for processor N.
Update After Read	When a value written by processor N is observed by a different processor, then no processor can later read any values written earlier by N.
Update After Read for Others	Finalizes the ordering of committed stores from different processors by tracking the order in which they are observed.
Update Add to Store Queue	An update for TSO to make the data structure more efficient
Update After Read TSO	Any store (to any address) by processor N is completed once a later value stored by processor N is observed.
Update After Store Time Based	Rough assumption that a store initiated some time after a previous store will commit later. “Some time” is a parameter that can be adjusted in the case of false positives.
Update After Sync Store	Upon Completion of a synchronous store operation, set the known commit time
Update Remove Read	Remove or garbage collect reads which no longer have relevant information for the scoreboard
Update After Read TCC	Tracks whether a given TCC read *could* lead to a violation
Update TCC	When in TCC mode, save operations to a special structure
Update After TCC Commit	Upon TCC Commit, invalidate earlier stores by other processors, check for violations

a histogram of the number of possibly allowed values for every simulated load in the test. Figure 5.4.d shows the results for a real application, in which there is very little uncertainty. This example shows two things. First, the scoreboard only occasionally needs to maintain more than one value, which shows the scoreboard is essentially equivalent to a deterministic checker in this case. Second, the test is not really stressing the design, because it is not inducing all the arbitration and conflict cases that could arise. Figure 5.4.a shows that for a random test, limited to 10 addresses for all 32 processors, the average uncertainty is 35 values and never exceeds 81. Figures 5.4.b and 5.4.c show that as there is less contention for the addresses there is also less uncertainty.

One conclusion we draw from Figure 5.4 is that the size of the set of possibly correct values is always bounded, even when the test focuses on a very small address range. In fact, for a self-checking diagnostic such as matrix multiplication (Figure 5.4.d), a relaxed scoreboard behaves almost identically to a golden reference model. In addition, it is important to note how much more stressful a random test with true sharing is (Figure 5.4.a) in comparison to a deterministic self-checking diagnostic (Figure 5.4.d). The latter rarely induced any races. This emphasizes the initial motivation for creating an end-to-end reference model that can be used with random tests, for a more efficient verification environment.

5.4 Relaxed Scoreboard Conclusions

The advantages of having a scoreboard or a golden model for validation are well known, as are the difficulties of creating this model. One approach to mitigate these difficulties is to allow a degree of flexibility in the reference model. Leveraging this flexibility and non-determinism to construct a *relaxed scoreboard*—a reference model that tracks a tight set of possible "right" answers and is therefore decoupled from implementation decisions—greatly simplifies the construction of the model. Since the possible set is almost always small, it does not change its effectiveness in finding errors in the design implementation.

The relaxed scoreboard creates a good framework for building chip multiprocessor memory checkers, since one can incrementally convert memory ordering or protocol properties into update rules for the scoreboard. A user creating a new protocol can include or exclude already existing rules, and easily add new checks and updates for their own protocol. This modularity is demonstrated by the Smart Memories verification effort, where the relaxed scoreboard was very effective in detecting errors in the implementation of multiple protocols.

Chapter 6

Conclusions & Future Work

This thesis has described a method for building hardware implementations of the complex memory protocols for modern multiprocessor systems. To enable the flexibility required to experiment with new techniques, we learned from the experience of building hardware programmable systems and moved instead towards a system that is "programmed" before tapeout. We defined an abstract hierarchical architecture, and constructed a flexible abstract template for a protocol controller between coherent caches. This template had an overall fixed structure composed of many smaller templates which were programmable versions of look up tables, TCAMs, memories, and more. An entire memory system is composed of several of these independent controllers, so our template also includes the interfaces between them in a separate description.

The resulting parameterized system is extremely complicated and detailed, which makes manually generating the parameters an extraordinarily difficult task. We therefore introduced SLAMM, a higher level language for generating the needed configuration for a desired memory protocol. SLAMM allows programming in C-like constructs, then our compiler extracts the necessary information from the specification to generate the XML format required by our template to configure it. The SLAMM compiler works well with other tools that configure other aspects of the overall system, so there is no problem with a designer adjusting other parameters of the system. These could include parameters for sizing, optimizations, and the number of units and the hardware structures which connect them.

We predicted and planned for the fact that no matter how powerful the SLAMM system is, inevitably there will be some blocks that it will need to interface to that were are not controlled by SLAMM. Therefore, there has to be a good mechanism for translating the

SLAMM system's internal decisions back to higher level concepts to allow designers to connect into the generated system. In addition, these mechanisms provide the possibility to debug the generated system or specification if there are errors in the resulting hardware. Concretely, this could be as simple as maintaining the information about enumeration mappings from higher-level names to values, rather than discarding the name information once the decision has been made.

This platform has the capability to allow modification of existing protocols to leverage modern protocols like those discussed in Chapter 2. The generic hardware template we have described is very good at implementing the control flow, as shown by the re-use of the structure in various controllers of our current system. The caveat here is that any new protocol will inevitably require the addition of some new hardware templates or blocks that didn't exist before. However, with our previous experience in building flexible systems, we expected this and planned for it by allowing clean ways to interface with them. To illustrate this point we briefly touch on a few of the protocols we mentioned earlier and consider how they could be described and implemented with this system.

ZCache [34] is a variation on cache coherency that aims to decouple ways and associativity. Its ways are mapped using a variety of hash functions to prevent lines from colliding across multiple ways. When looking at this from a SLAMM programming perspective, this means that the function inside of the Memory Block (cache) for doing this mapping is different internally. SLAMM does not describe the Tag mapping; it relies on a `lookupByAddress(Address)` primitive. By default, the hardware implements the `slamm_state` hardware which just does a basic lookup, but this hardware could be swapped out with a hardware implementation of the ZCache way mapping functionality.

To increase associativity, ZCache uses a sophisticated eviction candidate selection algorithm. It determines which eviction candidate is optimal from a large set. Using the multiple way functions, it creates and traverses a tree of eviction candidates. While a single line may map into only a few ways (such as three), the possible eviction candidates grows exponentially, so the best candidate can be selected from a much larger group. The result is that once the eviction candidate is selected, several data moves must be made to swap data around to their new homes. While traversing the tree to find the best eviction candidate fits well inside the function primitive (where eviction candidate selection currently resides), the process of moving data is more complex. SLAMM can describe this data movement as a series of transient states and associated internal messages.

Another of our modern memory system examples was the DataSafe [9] architecture. This introduced interface of Secure Data Compartments (SDC) managed by the hardware, special processor instructions to track them, and tag bits which are stored along with the data in the compartments in memory. DataSafe adds special off-core tracking structures, the `sdc_list` and `mem_map`. These are accessed explicitly by DataSafe-specific opcodes, for example `sdc_add` and `sdc_del` which add and delete elements from the hardware SDC list. Our SLAMM-programmed system template could incorporate these structures in a variety of ways. They could be incorporated into a memory block to change its behavior when accessed, or they could be described as entirely different memory blocks that must be accessed in addition to the cache (eg, processor checks the sdc_list before accessing the cache).

Our current whole system template contains a request network (Figure 3.3) which already supports an address map structure to determine which memory block to send an operation to, based on the address. For example, it would send the access to the instruction cache or the data cache depending on the address/processor port it comes out of (this can also depend on the operation). With DataSafe, loads to the SDC address region of memory are treated differently, but the SDC list is dynamic so this region is not fixed. Looking up whether a block is in this special region can be done if the SDC list is implemented as a specialized memory block or sophisticated state tracking structure.

Overall, when mapping new protocols onto our current system, we note that SLAMM focused on describing actions which result in sending messages out on wires to other state machines. SLAMM also addressed the issue of pulling in data from a variety of sources to send out on these interfaces, and automatically making sure that the necessary data is physically in place in order to be sent out. For other actions, SLAMM makes use of a clean interface of function calls with known inputs, and using the same methodology as sending out messages to provide the input data to the function input fields. Special function names (`get*`) allow the automatic use of prewritten hardware primitives, without preventing a user from writing their own.

Looking forward, there are two clear research directions to explore using this system as a starting point. First, we can use the SLAMM language in its current form and focus on creating a better hardware design output. Second, we can use the fact that we know how to generate hardware from a SLAMM specification, and come up with an even simpler, higher level specification that compiles down into a SLAMM specification.

First we can consider directions towards a more efficient hardware design. We have created a system that works, where the SLAMM compiler does some rudimentary optimization in the configuration for the hardware, but there is much more that it could do. The compiler could be enhanced to execute additional optimization passes, both at a local level or at a more global level. One example is the an optimized mapping between enumeration names and values. While enumerating states in SLAMM is straightforward, SLAMM does not provide an intrinsic way of describing a mapping between bits and states. An optimal mapping could exist which would greatly reduce the lines of code needed in the many TCAM structures. For example, an optimizing compiler could recognize that various STORE-like messages result in a similar set of actions, distinct from the actions taken from LOAD-like messages. It could then assign message type enumerations in such a way to make the comparisons in the TCAM more efficient.

We can also leverage the power of the generator's ability to take input from other specification tools to make the hardware more efficient. The generic template we described for a controller could be extended to support pipelining internally, similar to the protocol controller in Smart Memories. While this doesn't change the SLAMM specification of the protocol, it allows parameters such as pipeline depth to be chosen with optimization techniques such as those described by Azizi et al [6, 7]. Similar techniques can determine the optimal number of units or the structure of the interconnect.

The second direction for future work is to make it easier to specify protocols: SLAMM specifications are still quite detailed and require a good deal of user consideration of the system interactions. For example, SLAMM does not infer any intermediate states. When sending messages and waiting for a reply, the user must specify Wait states for each sort of message sent out. If multiple requests are sent out in parallel and then the replies are gathered before moving to the next state, the user has to carefully consider the possible orderings of the replies to make sure the appropriate intermediate states and transitions are all in place. While these interactions are complicated to describe, they follow similar patterns of request-response, scatter-gather. Future research can focus on ways on reducing this complexity by automatically mapping the protocol's true states (specified in an even higher level language) onto intermediate SLAMM states, with the associated transitions and messages.

Overall, our SLAMM-driven system serves as a good foundation on which future memory system generators can be built. The hardware is generic, programmable, and allows clean

interaction with custom hardware blocks, and provides many opportunities for expansion. The specification language significantly reduces the complexity of generating the required configuration data. Together they greatly reduce the effort required to create new memory systems.

Appendix A

Relaxed Scoreboard Details

A.1 Introduction

"The Relaxed Scoreboard" is a verification tool used for the Stanford Smart Memories Project, although whenever possible it tries to remain a general tool for verifying any memory system. The purpose of The Relaxed Scoreboard is to verify memory systems in which there may be many "right" answers, such as in a shared memory multiprocessor system.

In a single processor system:

Store A, 10 Load A

should always return 10. It is therefore simple to use an array or some other structure to represent the memory in software, and check against it as the program runs in hardware or hardware simulation. However, in a multiprocessor system, it is not so easy to determine what the correct answer for a Load should be, if many processors are writing to the same address. The Relaxed Scoreboard is an attempt to work around this problem.

A.2 Internal Structure of The Relaxed Scoreboard

The Relaxed Scoreboard is written in Vera. It is essentially an array of queues, with one queue for each address in the memory system. Because of masking, writes can be done on a byte level, so addresses are at the byte level. Thus, there is a queue for each byte address in memory. There are a few additional structures for handling things like Transactional Cache Consistency (TCC) and Total Store Ordering (TSO), and specifying address ranges

over which the Scoreboard should operate.

A.3 Relaxed Scoreboard Classes Summary

The relevant files and classes for The Relaxed Scoreboard are:

- Scoreboard.vr :

  The main Relaxed Scoreboard class which executes checks, updates, garbage collection, error reporting, and logging for the system

- Trace_trans.vr :

  Transaction class which is the element added to The Relaxed Scoreboard queues

- Trace_trans_Q.vr:

  A wrapper class around a verilog queue of Trace_trans to easily manage an array of queues.

- Scoreboard_aop_links.vr :

  Code to link The Relaxed Scoreboard into the rest of the verification environment: this is what actually makes calls to add transactions to The Relaxed Scoreboard.

- Scoreboard_Address.vr :

  A class just used as a container to specify an address range within the scoreboard (usually used to specify address ranges which the scoreboard should ignore).

- Store_Queue.vr :

  A class used for TSO to maintain pointers to a processor's store set within the main Scoreboard data structure.

- TCC_Transaction.vr :

  A class used for TCC to maintain pointers to a processor's R/W set during a TCC transaction within the main Scoreboard data structure, along with other TCC book-keeping information.

A.3.1 Trace_trans Class

The element of the queue is a Trace_trans object, which stores the basic information about a transaction. Most values are set with the AssignValues function before the transaction is added to the Scoreboard, but others are dynamically updated as the Scoreboard operates.

```
//  protected integer active;
//  protected integer my_ID; // the ID of the instance
//  protected string comment;
//  protected integer my\_creation\_time;

/*ID of sender entity (a processor or other part of the system),
 * as defined in
 * Quad\_def.vrh */
  integer senderID;

/*This is the physical byte address of this transaction,
 * translated by The Relaxed Scoreboard.*/
 bit [31:0] destinationAddr;

/*The write data if this is a write operation, or data returned on a load*/
  bit [7:0] data;

/*The global cycle when this transaction started*** */
 integer timeStart;

/*The global cycle when this transaction ended *** */
  integer timeEnd;

/*What operation was this (currently only Load or Store are supported*/
  integer operation;

/*For each CPU, what is the last clock cycle that a CPU can be
 * influenced by this transaction *** */
```

```
  bit [63:0] is_valid_for_cpu[*];

/* How many CPUs are currently active in the Scoreboard*/
 int num_cpus;

/* Indicates whether this is part of a non-committed TCC transaction*/
 bit transactional;

/* Indicates that this transaction is locked and should not
 * be Garbage Collected*/
 bit locked;
 }
```

***The timeStart, timeEnd, and is_valid_for_cpu[*] are dynamically adjusted as the scoreboard makes decisions and assumptions based on observed transactions.

A.3.2 Trace_trans_Q Class

The Trace_trans_Q is essentially a wrapper around a queue of Trace_trans structures. However, it also has some other functionality, such as being responsible for maintaining an idea of a locked address (meaning that an address has an outstanding operation pending and this queue should not be garbage collected).

```
class Trace_trans_Q
{
/*The queue which represents all the transactions to a current address.*/
  Trace_trans ttQ[$];

/* A lock for each processor in the design (if a processor is
 * using this address, it can lock it to prevent garbage collection. */
  integer locks[*];
 }
```

A.3.3 TCC_transaction Class

This class represents a TCC transaction. A transaction has numerous state flags and an associated list of Trace_trans which were executed under the transaction.

```
class TCC_Transaction
{
  /* Pointers to transactions completed under this TCC transaction*/
  Trace_trans ttQ[$];

  /*this represents a transaction which is underway*/
  bit valid;

  /* this represents a transaction which is committing*/
  bit committing;

  /* this represents whether a transaction CAN complete.
   * If a transaction is violated (according to the
   * scoreboard), it will be cleared. But, just because it is
   * 1 does not mean that this MUST commit.*/
  bit can_commit;

  /* this represents whether a transaction CAN be violated.
   * this means that it has read an address that was written
   * by another transaction. The ultimate violation will
   * happen depending on the order of commits. Just because this
   * flag is set does not mean that the transaction MUST violate.
   */
  bit can_violate;

  /*processor issuing this transaction*/
  integer proc;
}
```

A.3.4 Scoreboard Class

The Scoreboard class implements most of the functionality of the Relaxed Scoreboard as well as contains its major data structures. The functionality will be explained in later sections, here I just give an explanation of its data structures.

```
class Scoreboard extends GENERIC_master
{

/*Control variables */
  protected integer active;
  protected string my_agent_name;
  protected integer my_agent_ID;
  protected string my_prefix;
  protected integer _DEBUG_;

/*Count how many transactions were actually added to the scoreboard.
  protected integer transaction_counter;

/*semaphore for protecting the scoreboard when adding a transaction. */
  protected integer semID;

/**Queues**/

/* The main Data Structure: holds a Trace_trans_Q for each active
 * address in the Scoreboard.*/
  protected Trace_trans_Q  Trace_Q_Array[];

/* We do not use Vera's garbage collection of the Trace_trans_Q's,
 * so put allocated queues onto a freelist for re-use */
  protected Trace_trans_Q Trace_Q_Freelist[\$];

/* We do not use Vera's garbage collection of the Trace_trans,
 * so put allocated trans structures onto a freelist for re-use */
```

```
  protected Trace_trans Trace_trans_Freelist[$];

/*Structure for holding the addresses which the scoreboard should ignore
 * (these are specified on the command line) */
  protected Scoreboard_Address ignoreAddress;

/* Structure for holding the address which the scoreboard should
 * consider to be operating under a TCC memory model
 * (these are specified on the command line) */
  protected Scoreboard_Address tccAddress;

/* Array of TCC Transactions (one per pair of processors in the
 * Smart Memories implementation). */
  protected TCC_Transaction Tcc_Array[*];

/* Array of Pointers to current store set, for use with TSO
 * memory model. One per processor. */
  protected Store_Queue Store_Queues[*];

/* Lightweight structure to handle bookkeeping on addresses where
 * an entire Trace_trans structure is not needed, ie, addresses
 * for which there is no ambiguity. */

protected bit[8:0] BackupMem[];
...
}
```

A.4 Interfacing with The Relaxed Scoreboard

There are several ways in which transactions can be added to The Relaxed Scoreboard.

- **Transactions added by Scoreboard_aop_links**

  The common-case way for transactions to be added to The Relaxed Scoreboard is by the processor trace, in Scoreboard_aop_links.vr which monitors the signals coming

from each processor. When a load or store is detected (from either the data or instruction port), a transaction is sent to The Relaxed Scoreboard for each byte of data. Thus, if a masked data write only writes a single byte, only one transaction will be sent to The Relaxed Scoreboard. If a write affects the entire word of data, then four transactions will be added to The Relaxed Scoreboard.

- **Transactions added by the Test Environment**

  The test environment can add things to The Relaxed Scoreboard by using writes to the MainMem structure. This is mostly used when the environment is being initialized, so that matching values can be put into The Relaxed Scoreboard.

- **Transactions added by the Scoreboard**

  The Relaxed Scoreboard can add values to itself. For example, if a processor issues a load from an address that has never been written, The Relaxed Scoreboard can add an initial dummy store transaction before the load to handle this case (this is described in more detail in the Update section below).

A.5 Relaxed Scoreboard Operation

There are 3 main steps when a transaction is added to The Relaxed Scoreboard.

1. Address Translation
2. First, the CHECK step makes sure that this transaction was valid.
3. Next, the UPDATE step makes changes to The Relaxed Scoreboard, using the information learned from this transaction.

There are additional functions which interface with the scoreboard outside of these functions, both external and internal functions.

```
task Scoreboard::AddTransToSB(Trace_trans trac)
{
 /***************************************************************************
   * 1. Check the validity of the transaction.   **************************
  *************************************************************************
```

```
   * 2. Translate the address and add the transaction to the queue
   ***********************************************************************/
 /* Check for SPEC_CMD (TCC flags) and handle them if necessary*/
 /* translate the address : some opcodes do not require translation */
 /* Find the corresponding queue in the Trace_Q_Array.
  * If it does not exist, allocate one.*/
 /* Add the transaction to the correct queue. */
  /**********************************************************************
   * 3. Start the checking mechanism
   ***********************************************************************/

  /*HACK: set the transactional (TCC) flag here so CHECK can use it.*/
  /*log the transaction to the Scoreboard log file*/
  /* Execute CheckScoreboard -- CHECK step */
 /***********************************************************************
   * 4. Start invalidation of old transactions mechanism
   ***********************************************************************/
 /* Execute UpdateScoreboard  UPDATE step*/
}
```

A.5.1 Address Translation Step

The address translation is done using a TranslateAddress Vera function, which directly examines the segment table for each processor to translate the given address. This happens for each added transaction, which means that if the segment table is changed, The Relaxed Scoreboard can keep up.

```
/* TranslateAddress: translates the address from virtual to physical
 */
function bit[31:0] TranslateAddress(integer src_ID,
                                    bit[31:0] addr_in,
                                    integer op)
{
/*Translates the address if the 'op' requires it, by using the hardware
```

```
 * segment table, and returns the translated address.*/
}
```

A.5.2 The CHECK Step

Most of the work in checking is actually done by the Update Step. Checking is currently fairly straightforward. Currently, the only function used is CheckLastReceived. This also incorporates some extra complexity for handling TCC operations.

This does the following for a transaction issued by processor N:

1. If the last transaction is not a load, return PASS.
2. For all earlier transactions t to this address, starting with the latest:
   (a) If t was a store
   (b) if t's is_valid_for_cpu[N] is greater than the last transaction's start time
   (c) if t's data matches the last transaction's data
   (d) return PASS.
3. If all the earlier transactions are checked and there is no match, then
   (a) If this is the first transaction to this address, then it is allowed. return PASS
   (b) If this is the first read to this address, then this is also allowed. return PASS
4. return FAIL

CheckLastReceived

```
function integer Scoreboard::CheckLastReceived(Trace_trans last_trans,
                                               Trace_trans_Q added_to_q){
  integer jdx = 0;
  integer q_size = 0;


  bit[31:0] last_addr = last_trans.destinationAddr;
  Trace_trans trac_to_compare_with;
  integer value_was_not_stored;
```

```
  /*keep track if this processor has done a store here,
   * in case this is the first load. If the processor
   * has not done a store and this is the first load,
   * then allow it.
   */

  bit this_proc_stored_here = 0;

  // Initial return value is 1 (as in data not found)

  if (last_trans.operation is not a  LOAD)
     return PASS;
  for (all earlier transactions t to this address)
    if  (t was a store
    AND  t's is_valid_for_cpu[N] > the last_trans's start time
    AND  t's data matches last_trans's data)
      return PASS;

  if (last_trans was transactional and read a 0)
  return PASS;

  ERROR( Matching value was not found for the load,
         print all values the Scoreboard would have
         considered as correct).

 return FAIL;

}
```

CheckCountValidXactions

This check is simply for bookkeeping, to give an idea of how much ambiguity there is in the scoreboard for each load. The return value is always PASS, this check simply updates

and logs some internal counter information. This check was used to generate graphs for the paper about uncertainty window sizes.

```
function integer Scoreboard::CheckCountValidXactions(Trace_trans last_trans,
                                                     Trace_trans_Q added_to_q)
{

 integer num_valid_values = 0;
 if (last_trans.operation was not a load)
  return PASS;

 for (all earlier transactions t to this address)
    if  (t was a store
    AND  t's is_valid_for_cpu[N]  is greater than the last_trans's start time
    AND  t's data matches last_trans's data)
            num_valid_values = num_valid_values +1;

   LOG (last_trans.destinationAddr, num_valid_values);

  return PASS;
}
```

A.5.3 The UPDATE Step

There are many updates to The Relaxed Scoreboard. Some are designed for TCC or TSO specifically. TCC updates are run transparently to the caller (they simply do not get called because no transactions are identified as transactional) but TSO updates must be commented out/added manually currently.

UpdateSimple

If processor N does the following at the same address:

Store A Store B

Then any subsequent reads by processor N must return either B or something written by another processor, but not A. UpdateSimple is invoked when a store transaction is added

to The Relaxed Scoreboard. It will invalidate the previous store by the same processor, setting its is_valid_for_cpu[N] time to be the finish time of the current store. Because of inductance, all earlier stores will also be invalidated.

```
function Scoreboard::UpdateSimple(Trace_trans store_trans,
                                  Trace_trans_Q added_to_q)
{

  integer senderID = store_trans.senderID;
  bit [31:0] addr = store_trans.destinationAddr;
  integer q_size = 0;
  integer idx = 0;
  Trace_trans old_trans;

if (store_trans.operation is not a store)
  return;
for (all transactions t to this address)
  if (t is a store
     AND t.senderID == store_trans.senderID
    AND neither is transactional)
                  t.is_valid_for_cpu[N] = min(t.is_valid_for_cpu[N],
                                              store_trans.timeStart)

}
```

UpdateAfterRead

Suppose we have the following transactions, all to the same address:

```
Time ----------------->
p0: store A - store B - store C
p1: store D - store E - store F
p2: store G - store H - store I
```

Now, suppose we do a read with p0. Of course we know from the earlier check UpdateSimple that A and B are not acceptable. However, if the read returns I, then we know that I has committed and no earlier writes from p2 should be allowed for ANY processor. Therefore, this update will go through the transaction queue, and for any store by p2, it will invalidate p2's earlier stores for ALL processors, by setting its is_valid_for_cpu[i] to be the timeEnd of the current Load. It will actually do the minimum of the current is_valid_for_cpu[i] and timeEnd, in case the store has already been invalidated for some processors.

Note, that we can't really say anything if there were two stores of value 'I'. So, this update also makes sure that there is no ambiguity in the satisfying store before executing updates on the scoreboard.

TCC reads do not affect other processors so for transactional loads we skip this update.

```
function Scoreboard::UpdateAfterRead(Trace_trans read_trans,
                                     Trace_trans_Q added_to_q)
{

  Trace_trans trac_to_compare_with;
  bit found_match = 0;
  bit found_more_than_one_match = 1;
  integer matching_sender = 0;

  if (read_trans is not a load OR if read_trans is transactional)
   return ;

  //if this didn't come from a processor, return
  if (read_trans.senderID < 0  || read_trans.senderID > num_cpus-1){
    UpdateAfterRead = 0;
    return;
  }

  /*Using the same mechanism as CheckSimple, find a matching_store.
   * If there is no matching store, error out. If there is more
   * than one matching store, we can't do anything, so return.
```

```
   */

   /*since we have read this value, we know that it has definitely
    * committed. So we should update its commit time.
    */

    matching_store.timeEnd = min (matching_store.timeEnd,
                                  read_trans.timeEnd);
    /*now start searching from where you stopped and invalidate
     *the earlier stores for all processors*/

    for (all earlier stores t earlier than matching_store)
      for (each processor n)
        t.is_valid_for_cpu[n] = min(t.is_valid_for_cpu[n],
                                    read_trans.timeEnd);

}
```

UpdateAfterReadForOtherProcs

Suppose we have the following transactions, all to the same address:

```
   Time ----------------->
   p0: store A - store B - store C
   p1: store D - store E - store F
   p2: store G - store H - store I
```

If pX reads 'C', then we know the value 'C' has already committed. If at some later time pX reads 'H', then we know that H committed after C. Therefore, we can invalidate 'C' at the time of the read 'H' completed.

This update does not make sense for TCC loads because they do not affect other processors.

```
function Scoreboard::UpdateAfterReadForOtherProcs(Trace_trans read_trans,
                                                   Trace_trans_Q added_to_q)
```

```
{

  if (read_trans is not a LOAD
  OR read_trans is transactional)
     return;

 /* Using the same mechanism as CheckSimple, find a matching_store.
  * If there is no matching store, error out. If there is more
  * than one matching store, we can't do anything, so return.
  */

  For (all transactions t to this address)
    if (t is the matching_store) continue;
    if (t is a STORE
  AND t.timeEnd < read_trans.timeStart
  AND t is not transactional)
  for (all processors n)
    t.is_valid_for_cpu[n] = min(t.is_valid_for_cpu[n],
                                              matching_store.timeEnd);
}
```

UpdateAddToStoreQueue

This update is only used in the TSO model. It adds outstanding stores to a supplementary data structure for each processor. The stores can be removed from this structure in various ways. This structure facilitates operations that need to update outstanding stores for a processor, such as memory barriers or UpdateAfterReadTSO (used only for TSO)

```
function Scoreboard::UpdateAddToStoreQueue(Trace_trans last_trans){
  if (last_trans.operation is not a store)
  return;

  Store_Queues[last_trans.senderID].push_back(last_trans);
}
```

UpdateAfterReadTSO

In the TSO memory model, there is a known total ordering of all stores to all addresses by a processor. Therefore, once the scoreboard can conclude that a single store has completed, any earlier stores by that processor to any address is known to also be completed.

```
function Scoreboard::UpdateAfterReadTSO(Trace_trans read_trans,
                                        Trace_trans_Q added_to_q){

  if (read_trans is not a LOAD)
    return;

  /*Using the same mechanism as CheckSimple, find a matching_store.
   * If there is no matching store, error out.
   * If there is more than one matching store, we can't do anything,
   * so return.
   */

  for (each store s in Store_Queues[matching_store.senderID])
    if (s.timeStart < matching_store.timeStart)
      s.timeEnd = min(s.timeEnd, matching_store.timeEnd);
}
```

UpdateAfterStoreTimeBased

Although the hardware makes no guarantees about how long operations will remain outstanding, we can make assumptions to limit uncertainty in the scoreboard. For example:

```
P0 : store X at time 0
P1: store Y at time t > MAX_STORE_TIME
```

We can conclude that X will commit before Y.

Of course, if these assumptions are violated then the scoreboard will give false positives on errors, but this means simply adjusting the MAX_STORE_TIME time limit on how long operations will remain outstanding. In our implementation this has a default value of 1500 cycles, but can be adjusted with a command line parameter for tests which are known

to have long commit times (eg, where there is a lot of contention on a limited number of addresses).

```
function Scoreboard::UpdateAfterStoreTimeBased(Trace_trans store_trans,
                                               Trace_trans_Q added_to_q) {
  if (store_trans.operation is not a non-blocking store
      OR store_trans is transactional)
    return;
  for (all transactions t to this address)
    if  (t is a non-blocking STORE
    AND  t.timeStart + MAX_STORE_TIME <= store_trans.timeStart)
      for (all CPUs n)
        t.is_valid_for_cpu[n] = min(t.is_valid_for_cpu[n],
                                    store_trans.timeStart
                                      + MAX_STORE_TIME);
}
```

UpdateAfterSynchStore and CompleteTransInSB

This Update is not actually called as part of the usual update chain. Instead, it is called from the CompleteTransInSB function, which is used by the scoreboard for synchronous operations (known completion times). The Scoreboard_aop_links first send the store as usual when it is first observed at the interface (in order to allow other loads to see it). Scoreboard_aop_links then waits until it sees that its processor is unstalled (the store has completed) before calling CompleteTransInSB.

```
task Scoreboard::CompleteTransInSB(Trace_trans trac){

  bit UpdateScoreboard_error = 1'b0;
  if (trac is not a blocking store)
    return;

  UpdateAfterSynchStore(trac);
}
```

This is a function that invalidates synch stores. Because a synch store is guaranteed to have already completed, then a second synch store to the same address will invalidate all earlier synch stores. Unfortunately, this doesn't QUITE work... some synch stores may complete close together. So this should simply set the completion time of a synch store. UpdateAfterRead should invalidate things that have already completed. For now we leave this with the MaxStoreTime condition in place.

```
function Scoreboard::UpdateAfterSynchStore(Trace_trans store_trans){

  if (store_trans is not a blocking store)
    return;
  for (all other transactions t to this address)
    if (t is a blocking store
    AND t.timeEnd + MAX_STORE_TIME < store_trans.timeEnd)
      for (all processors n)
        t.is_valid_for_cpu[n] = min(t.is_valid_for_cpu[n],
                                    store_trans.timeEnd);
}
```

UpdateConsiderReadStartTime

```
function UpdateConsiderReadStartTime(Trace_trans last_trans,
                                     Trace_trans_Q added_to_q);
```

UpdateRemoveRead

With most memory models there is no reason to store reads, because we save the information given to us by the read in the store transactions. So if the last operation was a read, just remove it from the queue. If it is not removed here, it will be removed by the garbage collector.

```
function Scoreboard::UpdateRemoveRead(Trace_trans last_trans,
                                      Trace_trans_Q added_to_q){
```

```
  /*leave transactional loads*/
  if (last_trans is not transactional
  AND last_trans is a load){

    remove last_trans from its queue and return it to the freelist.
}
```

UpdateAfterReadTCC

This check, relevant only for the TCC memory model, does violation checking on loads. Consider the following trace:

```
initial condition: Addr a = y
Time      P0        P1
  0    START      START
  1    store x  a      ...
  2    START_COMMIT    ...
  3    ...        load y  a
  4
```

At this point, the scoreboard knows only that P1 saw a value that COULD lead to a violation. Since the scoreboard tracks both whether a transaction could violate as well as if it should, the above is a case in which P1's transaction could violate but does not have to. Note that we do not bother to check the value returned in the check below, since different processors could store the same value (ie, $x == y$) but the hardware would not distinguish this case, so the scoreboard does not either.

```
function Scoreboard::UpdateAfterReadTCC(Trace_trans read_trans,
                                        Trace_trans_Q added_to_q) {
  if (read_trans is not transactional
  OR  read_trans is not a load)
    return;

  for (each transaction t to this address)
    if (t is transactional
```

```
    AND t is a store
    AND t.is_valid_for_cpu[read_trans.senderID] >= 0) /*committing store*/
    AND t.senderID != read_trans.senderID)
      TccArray[read_trans.senderID/CPUS_PER_TILE].can_violate = 1;
}
```

UpdateTCC

When transactions come on the processor interface, the scoreboard must internally decide whether they are transactional or not. This is based on the state of the SPEC_CMD flags that the scoreboard has observed, as well as the address ranges which are specified at the command line to be TCC addresses. When the scoreboard sees a memory transaction at the interface, it uses this update to handle TCC bookkeeping if appropriate.

```
function Scoreboard::UpdateTCC(Trace_trans last_trans,
                               Trace_trans_Q added_to_q){

  TCC_Transaction tcc;

  /*First, we only care about this if the sender is in a transaction*/
  if (!(Tcc_Array[last_trans.senderID/CPUS_PER_TILE].valid)){
     return;
  tcc = Tcc_Array[last_trans.senderID/CPUS_PER_TILE];

  if (last_trans.destinationAddr is not in the TCC coherent/buffered region)
    return;
  if (tcc is committing)
    ERROR(shouldn't see transactional loads or stores during commit).

  /*Set this as invalid for all other processors until after commit.*/
  for (all processors n)
  if (n != last_trans.senderID)
    last_trans.is_valid_for_cpu[n] = 0;
  /* Set this load as invalid for all procesors
```

```
   *  (use this field as a 'dirty' bit)*/
  if (last_trans is a Load)
    for (all processors n)
      last_trans.is_valid_for_cpu[n] = 0;

  /*add this transaction to the processor's transaction queue*/
  last_trans.transactional = 1;

  last_trans.locked = 1; /*don't allow gcing*/

  tcc.push_back(last_trans);
}
```

UpdateAfterTCCCommit

When a TCC transaction fully commits and we observe the special memory transaction END_COMMIT, then for each store in the transaction we must:

- Set its end time
- invalidate any stores with earlier end times, for all processors.
- Check for violations

Violation checking here is somewhat tricky. If we see a transactional load to this address, the simple assumption is that it has read a stale, non-committed value and should definitely violate. However, it may be that the load occurred after the START_COMMIT operation, in which case it is possible for the transactional load to have seen this committed data, which is correct behavior and should not lead to a violation. So, we must handle this case. What we do is to use dirty bits for transactional loads: if they occurred before the START_COMMIT flag was seen, then they must violate. If their dirty bits aren't set (see UpdateAfterTCC) then we also check the data returned by the load. If it matches the committed value here, then we allow the transaction to either violate or commit. If the data doesn't match, then we must also consider the case that the value was written by its own transaction (SW as well as SR). So we also check for that case.

```
function Scoreboard::UpdateAfterTCCCommit(Trace_trans store_trans){
```

```
if (store_trans is not a store)
  return;
store_trans.timeEnd = current time;
for (each other transaction t to this address)
  if (t is a store
  AND t is not transactional
  AND t.timeEnd < store_trans.timeEnd)
    for (each processor n)
      t.is_valid_for_cpu[n] = min(t.is_valid_for_cpu[n],
                                  store_trans.timeEnd);
  else if (t is a transactional load)
    /*invalidate the corresponding TCC transaction
     * - it has been violated!*/
     *IF the dirty bit is set! If it is not, then this load happened
     * AFTER TccCommitStart,
     * so it should be allowed.
     */
    if (t.senderID == store_trans.senderID)
      continue;

    if (! old_trans.is_valid_for_cpu[store_trans.senderID/CPUS_PER_TILE])
      /*not dirty, no need to violate automatically.
       *Check the value received.*/
      if (old_trans.data == store_trans.data)
        /*This one has already read the committed data. It's OK.*/
        continue;
      else
        /*need to consider whether this word was not really SR,
         * because it had already been SM.
         */
        bit spec_modified = 0;
        for (each transaction k to this address older than t)
```

```
          if (t.senderID == k.senderID
          AND k is a store
          AND k wrote the same data t read)
            spec_modified = 1;
            if (spec_modified)
              continue to next t;

         tcc = Tcc_Array[t.senderID/CPUS_PER_TILE];
         tcc.can_commit = 0;

    /*finally, there may be outstanding loads that have not been seen by the
     * scoreboard. Invalidate those as well.*/
    for ( each cpu n)
      /*if the address is locked for that CPU, and the corresponding
       * TCC transaction is
       * valid, mark it as possibly violated.
       */
      if (Trace_Q_Array[addr].isLockedFor (n))
        TCC_Transaction tcc = Tcc_Array[n/CPUS_PER_TILE];
        tcc.can_violate = 1;

}
```

A.5.4 Additional Scoreboard Functions

CompleteTransInSB

This function is called by Scoreboard_aop_links (see UpdateAfterSynchStore above).

SetMemBarInSB

Certain operations define a Memory Barrier as part of their functionality (eg, blocking loads and stores). Since these are known at certain cycles after operations begin, the Scoreboard_aop_links is responsible for defining memory barriers. Rather than using a full Trace_trans transaction structure, we use a simple function since most of the fields in that

case are irrelevant.

```
task Scoreboard::SetMemBarInSB(bit[63:0] cycle, integer senderID){

  lock Scoreboard;
  for (each address a in the Scoreboard)
    bit latest_for_this_sender = 1;
    for (each transaction t at address a)
      if (t.senderID = senderID AND t is a store)
        if (t is a blocking store and t.timeEnd == INFINITY)
          /* this is a sync-op that is still going, probably the one
           * that caused this membar. */
          latest_for_this_sender = 0;
          continue;

    if (t is transactional)
      continue;
    t.timeEnd = min (t.timeEnd, cycle);
    if (latest_for_this_sender)
      latest_for_this_sender = 0;
    else
      for (each processor n)
        t.is_valid_for_cpu[n] = min (t.is_valid_for_cpu[n], cycle)
   unlock Scoreboard;
}
```

A.5.5 Scoreboard TCC Tasks

When the scoreboard observes a memory transaction with the opcode SPEC_CMD, it does not go through the usual check and update steps. Instead, it uses the arguments of the SPEC_CMD command as flags indicating special operations, specifically for TCC. This section describes the functions called when the scoreboard observes these flags.

HandleSpecCmd

Called from AddTransToSB when the scoreboard sees that the operation is SPEC_CMD, a speculative command.

```
function bit Scoreboard::HandleSpecCmd(Trace_trans specTrans){

bit err = 0;
  case(specTrans.destinationAddr)
    START_TRANSACTION:
      err = HandleTccStart(specTrans);
    END_OVERHEAD_START_TRANSACTION:
      err = HandleTccStart(specTrans);
    START_COMMIT:
      err = HandleTccCommitStart(specTrans);
    END_COMMIT:
      err = HandleTccCommitEnd(specTrans);
    VIOLATE_TRANSACTION:
      err = HandleTccViolation(specTrans);
    END_TRANSACTION:
      err = HandleTccEnd(specTrans);
    END_COMMIT_END_TRANSACTION:
      err = HandleTccCommitEnd(specTrans);
      err |= HandleTccEnd(specTrans);

  if (err)
    ERROR(HandleSpecCmd failed)
}
```

Handle TccStart

This function initiates a transaction in the scoreboard. It will fail if the processor already has a transaction (because SM does not support nested transactions.) Otherwise, it simply sets the transaction to be VALID.

```
function bit Scoreboard::HandleTccStart(Trace_trans specTrans){
```

```
  if (Tcc_Array[specTrans.senderID/CPUS_PER_TILE].valid){
    ERROR(The TCC transaction was already valid in call to HandleTccStart.
          Probably     START_TRANSACTION spec_cmd was called twice before
          a VIOLATE_TRANSACTION or   COMMIT_TRANSACTION\n");
  Tcc_Array[specTrans.senderID/CPUS_PER_TILE].valid = 1;

  return 0;
}
```

HandleTccCommitStart

When the Scoreboard observes the COMMIT_START flag, then it means that all the committing stores in the transaction can be observed by other processors, and sets their start time.

```
function bit Scoreboard::HandleTccCommitStart(Trace_trans specTrans){
  TCC_Transaction tcc;
  integer ii, jj;
  tcc = Tcc_Array[specTrans.senderID/CPUS_PER_TILE];
  if (!tcc.valid)
  ERROR(Transaction needs to be valid in order for it to commit)
  /* If this is already committing, then this shouldn't get called
   * again. */
  if (tcc.committing)
      ERROR(Committing transactions can't begin to commit again)
  if (!tcc.can_commit)
     MSG(Doing nothing because this transaction is not expected to
         successfully commit.)
     return;
 /*This transaction may begin committing at this point.
  * so, we should set the valid times for the other procesors now.
  * but. this doesn't affect any other stores. One thing
  * to note is that there may be multiple stores to the
```

```
  * same address. We only want the first one (most_recent_store),
  * which is noted by seeing if this is valid for this CPU until
  * infinity.
  */
 for (each transaction t in tcc)
    bit most_recent_store = (t.is_valid_for_cpu[specTrans.senderID]
                              == INFINITY);
    /*it is possible to call start/endCommit more than once for a transaction*/
   if (!t.transactional)
     continue;

   /*This store really begins at this time.*/
   t.timeStart = specTrans.timeStart;
   t.timeEnd = INFINITY;
   if (most_recent_store)
     for (each processor n)
       t.is_valid_for_cpu[n] = INFINITY;
       for (each other transaction k at t.destinationAddr)
         if (k.transactional AND  k is a load)
           /*flag that the corresonpding transaction COULD be violated.*/
           Tcc_Array[k.senderID/CPUS_PER_TILE].can_violate = 1;
           /*Set a "dirty" bit.*/
           k.is_valid_for_cpu[t.senderID/CPUS_PER_TILE] = 1;

  tcc.committing = 1;
}
```

HandleTccCommitEnd

The purpose of this function is to handle the completion of a TCC transaction. The outcome of this is that:

- The committed stores should be visible to all processors (this already happened in the first part of commit. But now, any earlier stores are invalid for all processors).

- The loads should be removed (invalidated so the GC picks them up)
- Any loads made by another TCC transaction to any address which we are writing to need to be flagged as unable to commit. This actually happens in the UpdateAfterTCCCommit function.

```
function bit Scoreboard::HandleTccCommitEnd(Trace_trans specTrans){
  integer ii;
  TCC_Transaction tcc;
  tcc = Tcc_Array[specTrans.senderID/CPUS_PER_TILE];
  if (!tcc.committing || !tcc.can_commit)
    ERROR(Scoreboard thinks this transaction should have VIOLATED.)
  for (each transaction t in tcc)
    bit most_recent_store;
    /*it is possible that this transaction has already committed
     *  due to overflows. Skip it if so */
    if (!t.transactional)
      continue;

    most_recent_store = (t.is_valid_for_cpu[specTrans.senderID]
                          == INFINITY);
    if (most_recent_store AND t is a store){
      t.timeEnd = specTrans.timeStart;
      UpdateAfterTCCCommit(t);
  t.transactional = 0;
  tcc.committing = 0;
 }
```

HandleTccViolation

In a Tcc violation, the main function is to delete the transactions from the scoreboard. This is done by invalidating them and letting the GC take care of them.

```
function bit Scoreboard::HandleTccViolation(Trace_trans specTrans){
  TCC_Transaction tcc;
```

```
  tcc = Tcc_Array[specTrans.senderID/CPUS_PER_TILE];
  if (!tcc.valid){
    ERROR(VIOLATE called before START_TRANSACTION)
    /*There shouldn't be a violation for no reason.*/
    if (tcc.can_commit & !tcc.can_violate){
      ERROR(Performance Bug: Transaction violated unnecessarily
    /*Clear dirty bits*/
    if(tcc.committing){
      for (each transaction t in tcc)
        for (each other transaction k at t.destinationAddr)
          if (k is transactional load )
            k.is_valid_for_cpu[t.senderID/CPUS_PER_TILE] = 0;

  for (each transaction t in tcc)
    t.is_valid_for_cpu[specTrans.senderID] = 0;
    t.transactional = 0;
    t.locked = 0;

  tcc.reset();
}
```

HandleTccEnd

The purpose of this function is to handle the completion of a TCC Transaction. This marks the end of the transaction and retires the tcc structure. There may be multiple commit pairs before this.

```
function bit Scoreboard::HandleTccEnd(Trace_trans specTrans){
  TCC_Transaction tcc;
  tcc = Tcc_Array[specTrans.senderID/CPUS_PER_TILE];
  if (!tcc.valid)
    ERROR(END_TRANSACTION called before START_TRANSACTION)
  if (tcc.committing)
    ERROR(END_TRANSACTION called before END_COMMIT)
```

```
  for (each transaction t in tcc)
    t.locked = 0;
  tcc.reset();
}
```

A.6 Garbage Collection

Garbage collection is an integral part of the Scoreboard, as it is responsible for reclaiming parts of the data structure that are no longer considered valid or relevant to the current state of the design. Every 1000 cycles or so, the garbage collector task iterates over the entire scoreboard and removes any transactions whose latest valid_time is less than the current time. It bypasses any locked addresses or transactions.

```
task Scoreboard::GarbageCollector(){

  while (active){

    wait 1000 clock ticks or so

    lock Scoreboard;

    invalidate_time = current time;
    /*Remove any completed stores from store queues (TSO)*/
    for ( each store_q q)
      for (each transaction s in q)
        if (s.timeEnd < invalidate_time)
          remove s from q;

    for (each address a in the scoreboard)
      if (Trace_Q_Array[a] is locked)
        continue;
      for (each transaction t at address a)
        if ((t is a store
            AND max(t.is_valid_for_cpu[n]) <= invalidate_time)
```

```
          OR t is a load)
         AND t is not transactional
         AND t is not locked )
     remove t from address a
     reset t
     return t to the freelist.

     /*If there is only one, old, transaction at this address,
      use the smaller backup structure instead.*/
     if (there is only one transaction t at address a)
       if (t.timeStart + MaxStoreTime < invalidate_time){
         BackupMem[a] = t.data;
       return the queue at address a to the freelist.

  unlock Scoreboard ;
}
```

A.7 Scoreboard Output and Error Reporting

The Relaxed Scoreboard is currently activated in the test environment if the -sb option is used. The Relaxed Scoreboard creates an output file, Quad_Scoreboard.log, which shows The Relaxed Scoreboard's view of all the transactions that are added to it. Any errors thrown by The Relaxed Scoreboard generally cause the environment to FAIL and an error message will show up in Quad_Scoreboard.log and the QshimLog*.log file.

Appendix B

Full SLAMM Grammar

The full SLAMM grammar is provided below:

```
   file: decl_list
decl_list:  decls
decls: decl decls
     | NULL
decl:  MACHINE_DECL     ( ident pair_list ) { decl_list }
    |  ACTION_DECL      ( ident pair_list ) statement_list
    |  IN_PORT_DECL     ( ident , type , var pair_list ) statement_list
    |  OUT_PORT_DECL    ( ident , type , var pair_list ) SEMICOLON
    |  TRANSITION_DECL  ( ident_list , ident_list , ident pair_list ) ident_list
    |  TRANSITION_DECL  ( ident_list , ident_list         pair_list ) ident_list
    |  EXTERN_TYPE_DECL ( type pair_list ) SEMICOLON
    |  EXTERN_TYPE_DECL ( type pair_list ) { type_methods }
    |  GLOBAL_DECL      ( type pair_list ) { type_members }
    |  STRUCT_DECL      ( type pair_list ) { type_members }
    |  ENUM_DECL        ( type pair_list ) { type_enums   }
    |  type ident pair_list SEMICOLON
    |  type ident ( formal_param_list ) pair_list SEMICOLON
    |  void ident ( formal_param_list ) pair_list SEMICOLON
    |  type ident ( formal_param_list ) pair_list statement_list
    |  void ident ( formal_param_list ) pair_list statement_list
type_members: type_member type_members
```

```
            | NULL
type_member: type ident pair_list SEMICOLON
           | type ident ASSIGN expr SEMICOLON
type_methods: type_method type_methods
            | NULL
type_method: type_or_void ident ( type_list ) pair_list SEMICOLON
type_enums: type_enum type_enums
          | NULL
type_enum: ident pair_list SEMICOLON
type_list : types
          | NULL
types    : type , types
         | type
type: ident
void: VOID
type_or_void: type
    | void
formal_param_list : formal_params
                  | NULL
formal_params : formal_param , formal_params
              | formal_param
formal_param : type ident
ident: IDENT
ident_list: { idents }
          | ident
idents:  ident SEMICOLON idents
      |  ident , idents
      |  ident idents
      |  NULL
pair_list: , pairs
         | NULL
pairs    : pair , pairs
         | pair
```

```
pair     : ident = STRING
         | ident = ident
         | STRING
statement_list:  { statements }
statements: statement statements
          | NULL
expr_list:  expr , expr_list
         |  expr
         |  NULL
statement:  expr SEMICOLON
         |  expr ASSIGN expr SEMICOLON
         |  ENQUEUE      ( var , type pair_list ) statement_list
         |  PEEK         ( var , type ) statement_list
         |  COPY_HEAD    ( var , var pair_list ) SEMICOLON
         |  CHECK_ALLOCATE ( var ) SEMICOLON
         |  CHECK_STOP_SLOTS ( var , STRING , STRING )SEMICOLON
         |  if_statement
         |  RETURN expr SEMICOLON
if_statement: IF ( expr ) statement_list ELSE statement_list
            | IF ( expr ) statement_list
            | IF ( expr ) statement_list ELSE if_statement
expr:  var
    |  literal
    |  enumeration
    |  ident ( expr_list )
    |  THIS DOT var [ expr ] DOT var DOT ident ( expr_list )
    |  THIS DOT var [ expr ] DOT var DOT field
    |  CHIP [ expr ] DOT var [ expr ] DOT var DOT ident ( expr_list )
    |  CHIP [ expr ] DOT var [ expr ] DOT var DOT field
    |  expr DOT field
    |  expr DOT ident ( expr_list )
    |  type DOUBLE_COLON ident ( expr_list )
    |  expr [ expr_list ]
```

```
    |  expr STAR  expr
    |  expr SLASH expr
    |  expr PLUS  expr
    |  expr DASH  expr
    |  expr <   expr
    |  expr >   expr
    |  expr LE    expr
    |  expr GE    expr
    |  expr EQ    expr
    |  expr NE    expr
    |  expr AND   expr
    |  expr OR    expr
    |  expr RIGHTSHIFT expr
    |  expr LEFTSHIFT  expr
    |  ( expr )
literal: STRING
       | NUMBER
       | FLOATNUMBER
       | LIT_BOOL
enumeration: ident : ident
var: ident
field: ident
```

Bibliography

[1] STMicroelectronics. http://www.st.com/index.htm.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Computer, 29(12):66–76, 1996.

[3] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. Computer, 29(12):66–76, Dec 1996.

[4] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.H. Lim, K. Mackenzie, and D. Yeung. The mit alewife machine: architecture and performance. In Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on, pages 2–13, 1995.

[5] Russell R. Atkinson and Edward M. McCreight. The dragon processor. In Proceedings of the second international conference on Architectual support for programming languages and operating systems, ASPLOS II, pages 65–69, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[6] Omid Azizi, Aqeel Mahesri, Benjamin C. Lee, Sanjay Patel, and Mark Horowitz. Energy-Performance Tradeoffs in Processor Architecture and Circuit Design: A Marginal Cost Analysis. In ISCA '10: Proc. 37th Annual International Symposium on Computer Architecture. ACM, 2010.

[7] Omid Azizi, Aqeel Mahesri, John P. Stevenson, Sanjay Patel, and Mark Horowitz. An Integrated Framework for Joint Design Space Exploration of Microarchitecture and Circuits. In DATE '10: Proc. Conf. on Design, Automation and Test in Europe, pages 250–255, March 2010.

[8] JF Cantin, MH Lipasti, and JE Smith. The complexity of verifying memory coherence and consistency. Parallel and Distributed Systems, IEEE Transactions on, 16(7):663–671, 2005.

[9] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A software-hardware architecture for self-protecting data. In Proceedings of the 2012 ACM conference on Computer and communications security, CCS '12, pages 14–27, New York, NY, USA, 2012. ACM.

[10] Yu-Yuan Chen and Ruby B. Lee. Datamoat: Architectural support for self-protecting data. Princeton University Department of Electrical Engineering Technical Report CE-L2011-002 (updated June 1, 2011), Feb. 10, 2011 2011.

[11] David Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P. Stevenson, and Omid Azizi. HICAMP: architectural support for efficient concurrency-safe shared structured data access. In Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII, pages 287–300, New York, NY, USA, 2012. ACM.

[12] A. Firoozshahian, A. Solomatnikov, O. Shacham, Z. Asgar, S. Richardson, C. Kozyrakis, and M. Horowitz. A Memory System Design Framework: Creating Smart Memories. In ISCA '09: Proc. 36th Annual International Symposium on Computer Architecture, pages 406–417. ACM, 2009.

[13] E. Gibbons, P.B.; Korach. The complexity of sequential consistency. Symposium on Parallel and Distributed Processing, pages 317–325, 1992.

[14] P.B. Gibbons and E. Korach. Testing Shared Memories. SIAM Journal on Computing, 26(4):1208–1244, 1997.

[15] R.E. Gonzalez. Xtensa: a configurable and extensible processor. Micro, IEEE, 20(2):60–70, Mar/Apr 2000.

[16] Michael Gschwind. Chip multiprocessing and the cell broadband engine. In CF '06: Proceedings of the 3rd conference on Computing frontiers, pages 1–8. ACM, 2006.

[17] L. Hammond, B.A. Hubbert, M. Siu, M.K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. IEEE MICRO, pages 71–84, 2000.

[18] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, and K. P Manohar. Transactional memory coherence and consistency. In International Symposium on Computer Architecture (ISCA '04), page 102, 2004.

[19] Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, and Juin-Yeu Joseph Lu. TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model. In ISCA '04: Proc. 31st Annual International Symposium on Computer Architecture, page 114. ACM, 2004.

[20] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of message passing and shared memory in the stanford flash multiprocessor. In Proceedings of the sixth international conference on Architectural support for programming languages and operating systems, ASPLOS-VI, pages 38–50, New York, NY, USA, 1994. ACM.

[21] John L. Hennessey and David A. Patterson. Computer Architecture, a Quantitative Approach. Morgan Kaufman, 1996.

[22] M.D. Hill. Multiprocessors should support simple memory consistency models. Computer, 31(8):28–34, Aug 1998.

[23] Ujval J. Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The imagine stream processor. Computer Design, International Conference on, 0:282, 2002.

[24] K. Kelley, M. Wachs, J. Stevenson, S. Richardson, and M. Horowitz. Removing overhead from high-level interfaces. In Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE, pages 783–789, 2012.

[25] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput., 28(9):690–691, September 1979.

[26] James R. Larus and Ravi Rajwar. Transactional memory. Synthesis Lectures on Computer Architecture, 1(1):1–226, 2006.

[27] James Laudon and Daniel Lenoski. The sgi origin: a ccnuma highly scalable server. In Proceedings of the 24th annual international symposium on Computer architecture, ISCA '97, pages 241–251, New York, NY, USA, 1997. ACM.

[28] K. Mai, T. Paaske, N. Jayasena, R. Ho, W.J. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. Computer Architecture, 2000. Proceedings of the 27th International Symposium on, pages 161–171, 2000.

[29] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. SIGARCH Comput. Archit. News, 33:92–99, November 2005.

[30] Paul E. McKenney. Memory ordering in modern microprocessors, part I and II. Linux J., 2005(136/7), 2005.

[31] John Owens. Streaming architectures and technology trends. In SIGGRAPH '05: ACM SIGGRAPH 2005 Courses, page 9, New York, NY, USA, 2005. ACM.

[32] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. SIGARCH Comput. Archit. News, 12(3):348–354, January 1984.

[33] A. Saha, N. Malik, B. O'Krafka, J. Lin, R. Raghavan, and U. Shamsi. A simulation-based approach to architectural verification of multiprocessor systems. Computers and Communications, 1995. Conference Proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference on, pages 34–37, Mar 1995.

[34] Daniel Sanchez and Christos Kozyrakis. The zcache: Decoupling ways and associativity. In MICRO'10, pages 187–198, 2010.

[35] Ofer Shacham. Chip Multiprocessor Generator: Automatic Generation of Custom and Heterogeneous Compute Platforms. Phd thesis, Stanford University, Stanford, CA, 2011.

[36] Ofer Shacham, Sameh Galal, Sabarish Sankaranarayanan, Megan Wachs, John Brunhaver, Artem Vassiliev, Andrew Danowitz, Wajahat Qadeer, Stephen Richardson, and Mark Horowitz. Avoiding game over: bringing design to the next level. In Proceedings of the 49th Annual Design Automation Conference, DAC '12, pages 623–629, New York, NY, USA, 2012. ACM.

[37] Ofer Shacham, Megan Wachs, Alex Solomatnikov, Amin Firoozshahian, Stephen Richardson, and Mark Horowitz. Verification of Chip Multiprocessor Memory Systems Using A Relaxed Scoreboard. In MICRO '08: Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture, pages 294–305, 2008.

[38] PradeepS. Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. In Michel Dubois and Shreekant Thakkar, editors, Scalable Shared Memory Multiprocessors, pages 25–41. Springer US, 1992.

[39] Alex Solomatnikov, Amin Firoozshahian, Wajahat Qadeer, Ofer Shacham, Kyle Kelley, Zain Asgar, Megan Wachs, Rehan Hameed, and Mark Horowitz. Chip multi-processor generator. In Proceedings of the 44th annual Design Automation Conference, DAC '07, pages 262–263, New York, NY, USA, 2007. ACM.

[40] Alex Solomatnikov, Amin Firoozshahian, Ofer Shacham, Zain Asgar, Megan Wachs, Wajahat Qadeer, Stephen Richardson, and Mark Horowitz. Using a configurable processor generator for computer architecture prototyping. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 358–369, 2009.

[41] Daniel J. Sorin, Manoj Plakal, Anne E. Condon, Mark D. Hill, Milo M. K. Martin, and David A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. IEEE Trans. Parallel Distrib. Syst., 13:556–578, June 2002.

[42] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The stampede approach to thread-level speculation. ACM Trans. Comput. Syst., 23(3):253–300, 2005.

[43] M. Talupur and M.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.

[44] M. Wachs, O. Shacham, Z. Asgar, A. Firoozshahian, S. Richardson, and M. Horowitz. Bringing up a chip on the cheap. Design Test of Computers, IEEE, 29(6):57–65, 2012.

[45] Zhenghong Wang. Information Leakage Due to Cache and Processor Architectures. Phd thesis, Princeton University, Princeton, NJ, 2012.

[46] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06), pages 473–482, December 2006.

[47] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In Proceedings of the 34th annual international symposium on Computer architecture, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.

[48] M.V. Wilkes. Slave memories and dynamic storage allocation. Electronic Computers, IEEE Transactions on, EC-14(2):270–271, 1965.