Simulation of CACHET on a Multiprocessor Computer

by

Andrew Michael Sarkar

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

at the
Massachusetts Institute of Technology

May 22, 2000

© 2000 Andrew M. Sarkar. All rights reserved.
The author hereby grants M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author__________________________
Department of Electrical and Computer Science
May 22, 2000

Certified by__________________________
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by__________________________
Arthur C. Smith
Chairman, Committee on Graduate Students
Simulation of CACHET on a Multiprocessor Computer

by

Andrew Michael Sarkar

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

A Distributed Shared Memory (DSM) system implements a coherent view of
global memory by having a number of sites connected to a common memory, such
that sites running programs in parallel can exchange shared variables and utilize
shared data to synchronize their programs with respect to one another. To
increase performance, a semantically invisible cache is attached to each site, which
allows shared variables to be in multiple sites’ caches simultaneously. The set of
rules governing cache behavior, called a cache coherence protocol, is one of the
most important factors regarding the complexity and overall performance of the
system. A cache coherence protocol must support a memory model, and the
Commit-Reconcile & Fences (CRF) memory model has been chosen because it
captures both ease of implementability and ease of programming. Cachet, a newly
developed cache coherence protocol to implement CRF, is composed of three
microprotocols, each of which has been optimized for particular memory access
patterns. The work of this thesis expands the understanding of Cachet, by
exploring how well it can be implemented in hardware. The high-level description
of Cachet is taken and rewritten to be suitable for input, as specified by TRSpec, to
the Term Rewriting Architecture Compiler (TRAC). During the rule translation,
we used the two-stage Imperative & Directive methodology, and the translated
rules go through TRAC to give Verilog RTL output. This Verilog RTL can then be
simulated and synthesized by commercial tools, as the next steps toward
implementing Cachet in hardware.

Thesis Supervisor: Arvind
Title: Professor of Electrical Engineering and Computer Science
Acknowledgments

First and foremost, I would like to thank my thesis advisor, Professor Arvind, for allowing me to do this interesting research and his continued support and encouragement. He has been extremely supportive through my difficulties in doing research, and his direction and support has helped to drive me to a goal long pursued.

There are, of course, many people who should receive credit for helping me through the work for my thesis. Although Xiaowei Shen graduated before I did, I knew him from working with him in Professor Arvind’s research group, as well as 6.826. His work provided the backbone on which I elaborate and communications with him have been helpful, even if he is in another state most of the time. What would we do without email?

I would like to thank fellow graduate student and Ph.D. researcher James Hoe for his guidance and help regarding TRAC and issues of Verilog simulation. He graciously answered many small and big questions that I asked of him, and my discussions with him have proven rewarding.

Another researcher deserving credit is Dan Rosenband, who assisted me in learning and beginning to understand Verilog. Even though I didn’t have time to move forward with the Verilog Simulation part of my thesis, it was a useful process for me to have undergone. Hopefully what I learned, will stay with me and help me to think critically for important project through the coming years of life.

I want to thank my office-mates, Mieszko Lis and Michael Sung, for putting up with me and helping me. Mieszko would help me numerous times with many questions that I had, and it didn’t take him long to answer me, but it sure saved me a lot of work. Thanks!

I would also like to thank Anne Hunter for her administrative excellence and the great help that she has rendered me over my many years. With her help I was finally able to complete the little things and the big things that stood in my way to get the degree.

Lastly, I would like to thank my family for helping me reach this milestone. I thank my parents for giving me the support and opportunity to explore my potential and fulfill my aspirations. It was a help to have Mom come down and live in Boston for my last term, and I appreciate how she put her life on hold for a few weeks to help me get this done. I would like to thank my brothers for being less annoying than they were in my childhood. But, seriously, it has been helpful for me to know I have a good relationship with my brothers and parents; strong roots will lead to a strong tree.

Andrew Sarkar
Massachusetts Institute of Technology
Cambridge, Massachusetts
May 22, 2000
# Table of Contents

Chapter 1 Introduction.............................................................................9

Chapter 2 Background & Literature Review..............................................13
  2.1 The Commit-Reconcile & Fences Memory Model.........................13
  2.2 The Cachet Cache Coherence Protocol......................................14
  2.3 Using TRSs to Design and Verify Processors..........................14
  2.4 Synthesis of Cache Coherence Protocols..............................15

Chapter 3 Cachet: A Cache Coherence Protocol for CRF....................17
  3.1 Description of Cachet..........................................................17
  3.2 Description of Cachet-Base................................................19
  3.3 Description of Cachet-WriterPush.......................................22
  3.4 Description of Cachet-Migratory.........................................27

Chapter 4 Synthesis of CACHET Microprotocols.............................31
  4.1 Signaling Convention..........................................................31
  4.2 Memory Controller State Definition....................................34
  4.3 Sache Controller State Definition.......................................36
  4.4 Cachet-Base........................................................................38
    4.4.1 Sache Controller of Cachet-Base...................................39
    4.4.2 Memory Controller of Cachet-Base...............................41
  4.5 Cachet-WriterPush.............................................................41
    4.5.1 Sache Controller of Cachet-WriterPush..........................41
    4.5.2 Memory Controller of Cachet-WriterPush.......................43
  4.6 Cachet-Migratory..................................................................45
    4.6.1 Sache Controller of Cachet-Migratory...........................46
    4.6.2 Memory Controller of Cachet-Migratory.........................47

Chapter 5 Conclusion..........................................................................49

Appendices

Appendix A: Implementable Protocol of Cachet-Base..........................51
Appendix B: Implementable Protocol of Cachet-WriterPush.................53
Appendix C: Implementable Protocol of Cachet-Migratory..................55
Appendix D: State Descriptions for Cachet....................................57
Appendix E: Computer Readable TRS of Cachet-Base.......................59
Appendix F: Computer Readable TRS of Cachet-WriterPush..............63
Appendix G: Computer Readable TRS of Cachet-Migratory..............69

Bibliography.........................................................................................73
List of Figures

Figure 1.1 Model of Cachet Synthesis ................................................................. 12

Figure 3.1 Interfaces for the Sache Controller and Memory Controller .......... 19
Figure 3.2 Directive Sache Controller Protocol of Cachet-Base .................. 21
Figure 3.3 Imperative Sache Controller Protocol of Cachet-Base ............... 21
Figure 3.4 Imperative Memory Controller Protocol of Cachet-Base ............. 22
Figure 3.5 Directive Sache Controller Protocol of Cachet-WriterPush .......... 25
Figure 3.6 Imperative Sache Controller Protocol of Cachet-WriterPush .......... 26
Figure 3.7 Imperative Memory Controller Protocol of Cachet-WriterPush ....... 26
Figure 3.8 Directive Sache Controller Protocol of Cachet-Migratory ............ 28
Figure 3.9 Imperative Sache Controller Protocol of Cachet-Migratory ......... 28
Figure 3.10 Imperative Memory Controller Protocol of Cachet-Migratory ....... 29

Figure 4.1 Asynchronous Handshake Signaling Convention ......................... 32
Figure 4.2 Rule Example ................................................................................. 33
Figure 4.3 Synthesizable Components of the Four-Site System .................... 34
Figure 4.4 Semantic Configuration of Memory ............................................... 36
Figure 4.5 Semantic Configuration of Sache .................................................. 38

Figure 5.1 Size of RTL Code Generated ............................................................ 49
Chapter 1

Introduction

In modern multiprocessor computers, we wish to have a coherent view of global memory, such that sites running programs in parallel can exchange shared variables and utilize shared data to synchronize their programs with respect to one another. A Distributed Shared Memory system implements the abstraction of having a shared global memory by having a number of sites connected to a common memory. However, memory access commands are the slowest component of almost every user program, and become even slower when we have multiple processors, because then the data bus becomes a shared resource and access to it is limited due to usage of the bus by the other processors.

To increase performance, a semantically invisible cache is attached to each processor, which will store the memory locations that the program on that processor commonly uses. The technique of caching allows shared variables to be in multiple sites' caches simultaneously so as to reduce memory access latency time, while remaining semantically invisible. A semantically invisible cache means that these caches should be invisible to the user in terms of program behavior. The cache is meant to increase performance, but the program behavior should not be changed. Employing the technique of caching will improve the performance of programs in this Distributed Shared Memory system that utilize shared variables.

To remain semantically invisible, and yet accomplish the goal of increasing program performance, we must have a set of rules governing cache behavior, and we call such a set of rules a cache coherence protocol. When can a cache get a copy of a memory location from global memory? How does the cache write back a change to that memory location, such that it will be globally visible? These questions are addressed in the set of rules that govern cache behavior: the cache coherence protocol. Cache coherence protocols tend to be quite complex and error prone, especially when high performance is the goal.

The time to access first-level cache on a microprocessor chip is typically two orders of magnitude faster than the time to access off-chip main memory. This
difference is so great, that whenever we can use cache information rather than going to global memory, it would be highly desirable to do so. In order to get the benefit of this difference, we need to have the desired memory location in cache. The cache coherence protocol will determine whether or not the desired memory locations are available in cache at the appropriate time. Not all protocols are equal; with a poor protocol, for example one in which the cache always remains empty, we could spend just as much time doing memory accesses as when we had no caches at all. Having an efficient protocol is, therefore, critically important for improving performance.

An issue relevant to cache coherence protocols for a Distributed Shared Memory is the question of what memory model should be supported [1]. A memory model defines the semantics of memory access instructions. An ideal memory model would be semantically simple, so that it is easy to write programs in that model, and easily implementable, so that it would allow efficient and scalable implementations of that memory model. However, no such memory model exists; current memory models are strong on one issue, either easily implementable or semantically simple, while being weak on the other issue. Sequential Consistency [7] is easy for programmers to understand and use, but is not easily implementable. Commit-Reconcile & Fences (CRF) [9], allows various optimizations and is easily implementable, but it is difficult to understand, relative to Sequential Consistency.

Due to CRF's ease of implementation, it is the memory model chosen to underlie the work of this thesis, and will be further described below. CRF is intended for architects and compiler writers, rather than for high-level parallel programmers. It exposes instruction reordering and data replication at the Instruction Set Architecture level, and it permits aggressive cache coherence protocols because no operation involves more than one semantic cache. A novel and attractive feature of CRF is that many memory models can be expressed as restricted versions of CRF; thus programs written under those restrictive memory models can be translated to efficient CRF programs. Translations of programs written under memory models such as Sequential Consistency or Release Consistency into CRF programs are straightforward. CRF captures ease of implementability, while being able to run programs written in Sequential Consistency. Since it is easy to reason about program behavior in Sequential Consistency, we have captured both ease of implementation and ease of
A new cache coherence protocol to implement CRF, named Cachet, was recently developed [10]. Different types of parallel programs running in the CRF memory model will have various memory access patterns, and the different patterns have aspects that could be taken advantage of by a cache coherence protocol. Cachet is composed of three microprotocols, each of which has been optimized for a particular memory access pattern. Also, since Cachet implements the CRF memory model, it is automatically an implementation for all the memory models whose programs can be translated into CRF programs, notably Sequential Consistency and Release Consistency.

Term Rewriting Systems (TRSs) is chosen as the underlying formalism to specify and verify computer architectures and distributed protocols because cache coherence protocols can be extremely complicated, especially in the presence of various optimizations. Using TRSs to define the operational semantics of the CRF memory model allows each CRF program to have some well-defined operational behavior.

Recently, there has been great progress in hardware synthesis from TRSs [4]. James Hoe's Term Rewriting Architecture Compiler (TRAC) compiles high-level behavioral descriptions written in TRSs into Register Transfer Language (RTL), a subset of Verilog. This output can then be simulated and synthesized using commercial tools. This TRAC compiler enables a new hardware design framework that can match the ease of today's software program environment, by eliminating the involvement in the lower-level implementation tasks, and thus the time and effort to develop and debug hardware will be reduced. The work of this thesis expands the understanding of the Cachet protocol, by exploring how well it can be implemented in hardware using James Hoe's tool.

The process to create hardware of the cache coherence protocols of Cachet involves many steps. The full process required, to ultimately reach the state where we have microchips that implement Cachet, is diagrammed in Figure 1.1.

The thesis work presented here, takes a high-level description of Cachet, expressed in a TRS by Xiaowei Shen, and rewrites it to make it suitable for hardware synthesis using the TRAC compiler. The second step of the thesis work was to compile the TRS description, using the TRAC compiler. There are two additional steps that logically follow this work, which have not been undertaken, due to time limitations.
The first of these two steps would be to take the RTL output of the TRAC compiler, and send it to a Real Circuit Simulator, to simulate the running of these cache coherence protocols. The step after that would be to synthesize and eventually manufacture working hardware that runs these cache coherence protocols.

This work has shown that the rules can be translated and compiled, and the Verilog RTL output of that compiler is ready for the next steps: simulation, synthesis and manufacturing.
Chapter 2
Background & Literature Review

The majority of the background material for this thesis was derived from a number of papers which describe research done by the Computation Structures Group (CSG) of the Lab for Computer Science (LCS) at the Massachusetts Institute of Technology (MIT). Most of the material here has been taken from Xiaowei Shen's Ph.D. thesis, entitled "Design and Verification of Adaptive Cache Coherence Protocols" [8].

2.1 The Commit-Reconcile & Fences Memory Model

Instruction reordering and caching are prevalent features of modern computer systems. While they are transparent to uniprocessor machines, in multiprocessor computers they are anything but transparent. A whole area of research has sprung up to determine what model of memory should be presented to 1) the compiler writer, 2) the computer architect, or 3) the programmer. The first background paper is "Commit-Reconcile & Fences (CRF): A new memory model for Architects and Compiler Writers", authored by Xiaowei Shen, Arvind, and Larry Rudolph [9]. This paper describes the CRF Memory Model, a mechanism-oriented memory model, and defines it using algebraic rules. CRF is defined to be easily implementable by architects, as well as provide a stable target machine interface for compiler writers of high-level languages. Many existing memory models are reducible to CRF. The approach of CRF is to decompose the read and write instructions into finer-grain orthogonal operations. A traditional read operation is decomposed into a Reconcile and LoadL, which reconciles the cache data with the rest of the system, and then loads the data locally (if possible; if not, it goes to memory). A traditional write operation is decomposed into a StoreL and Commit, which stores the data locally, and then commits the data back to global memory to allow it to be readable from any other site. It exposes both data replication and instruction reordering at the programmer level.
2.2 The Cachet Cache Coherence Protocol

The second reference paper, "CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared Memory Systems", authored by Xiaowei Shen, Arvind and Larry Rudolph [10]. This paper explains Cachet, a protocol for implementing the CRF memory model, in terms that a person who is conversant in multiprocessors and caching, can easily understand. Cachet is composed of three microprotocols, each of which is optimized for a certain memory-access pattern. Cachet embodies adaptivity for both intra-protocol and inter-protocol, to achieve high performance under changing memory access patterns. It is presented in the context of the CRF memory model. Other weaker memory models are reducible to CRF, thus a protocol to implement CRF, such as Cachet, is automatically a protocol to implement any of those weaker memory models. The three microprotocols that make up Cachet are named Cachet-Base, Cache-WriterPush, and Cachet-Migratory. Cachet-Base is the simplest of the three microprotocols and can always be used as the default protocol. For example, if we run out of spaces to store state on the memory side, by switching to the Cachet-Base protocol, we can at least have a correct cache coherence protocol implementation in place. Indeed, Cachet-Base has as its main feature the fact that no state needs to be maintained at the memory side. Its secondary feature is its plain simplicity; it is easily understood. Cachet-WriterPush is a protocol that is excellent for the programs that do many more reads than writes (reads are very cheap in this protocol). Cachet-Migratory might have been called Cachet-Exclusive, for its main feature is that any memory location that is accessed by only one processor for a reasonable length of time becomes assigned to that processor. Reads and writes by that processor to that location are very cheap. However, if any other processor then needs that memory location, a great deal of work is incurred to migrate the data from the first processor to the newly requesting processor.

2.3 Using TRSs to Design and Verify Processors

The third background paper, is “Using Term Rewriting Systems to Design and Verify Processors", authored by Arvind and Xiaowei Shen [2]. This paper describes a way of using TRSs to describe micro-architectures and cache coherence protocols. It shows how a TRS can give a convenient and easy way to prove the
correctness of an implementation with respect to a specification. The paper illustrates how TRSs can be used in this way by giving the operational semantics of a simple RISC instruction set. Then a TRS is used to describe and implement one of the RISC instruction sets, on a micro-architecture that allows register renaming and speculative execution. The two TRSs are shown to be able to simulate each other, and thus the correctness of the proposed implementation is proven. This shows how TRSs can be used to verify that one implementation does in fact implement the specification of a less complex instruction set.

2.4 Synthesis of Cache Coherence Protocols


The TRS compiler, which is called TRAC, generates an RTL description in Verilog, which is described in "Hardware Synthesis from Term Rewriting Systems", authored by James Hoe and Arvind [6]. The paper also describes the TRS notation accepted by TRAC, which includes built-in integers, booleans, common arithmetic and logical operators, non-recursive algebraic types and a few abstract datatypes such as arrays and FIFO queues. Other user-defined abstract datatypes, which both sequential and combinational functionalities, can be included in synthesis by providing and interface declaration and its implementation. A TRS is specified by \( T \), a set of terms, and \( R \), a set of rewriting rules. A term is a collection of elements from \( E \), which is the set of symbols. One element in \( T \) is designated as the starting term, which corresponds to the initial state of the computation. The rewrite rules specify the allowed state transitions from a given state, and these rewrite rules are atomic: thus the entire original term is read AND the applicability of the rule is determined AND the new term is returned all in one step. From the initial state, each rule application produced a new term that represents the next state in the process of the computation.

The paper entitled "Scheduling and Synthesis of Operation-Centric Hardware Descriptions", authored by James C. Hoe and Arvind [5], describes a synthesis that compiles an operation-centric description into a state-centric synthesizable RTL description, by finding a consistent and efficient scheduling of the independently prescribed operations. Detailed synthesis algorithms are
presented in the context of an Abstract Transition System (ATS), which is a suitable abstract representation of operation-centric descriptions in general. For performance reasons, the synthesized implementation should carry out as many operations concurrently as possible, and yet still produce a behavior that is consistent with the atomic and sequential execution semantics of the original operation-centric specification. Although the semantics of an ATS require executions in sequential and atomic update steps, there could be underlying parallelism and the hardware implementation can exploit this to execute two transitions concurrently in one clock cycle.

The complete language accepted by TRAC is specified by TRSpec, as defined in chapter 1 of James Hoe's Ph.D. thesis [3]. TRAC gives us a new ways to design hardware that is as easy and simple as today's software programming environment. By eliminating human involvement in lower-level implementation tasks, the effort of developing and debugging is reduced.
Chapter 3
Cachet: A Cache Coherence Protocol for CRF

The work of this thesis is based on Xiaowei Shen's Cachet cache coherence protocol for CRF [8]. This chapter gives a brief description of Cachet because an understanding of Cachet is needed to appreciate the synthesis work that follows.

Shen introduced a two-stage design methodology called Imperative & Directive that separates the soundness and liveness aspects of the design process. Soundness is the aspect that ensures that the system exhibits only legal behaviors, permitted by the specification, while liveness ensures that the system takes desirable actions, eventually, to make progress. In simpler terms, this phrase has been coined to explain it: "Soundness makes sure nothing bad ever happens, while Liveness ensures that eventually something good will happen."

3.1 Description of Cachet

Cachet is an adaptive cache coherence protocol, consisting of three microprotocols, each of which is optimized for a specific pattern of memory access. Cachet thus provides enormous adaptivity for programs with different access patterns, due to its seamless integration of these three microprotocols. Cachet also allows write operations to be performed without exclusive ownership of the data, and thereby reduces the average latency of writes, as well as alleviating potential cache thrashing due to false sharing. These features make Cachet a more attractive cache coherence protocol than others, especially for scalable Distributed Shared Memory systems.

The scope for adaptive behavior within each microprotocol is based on voluntary rules, which can be triggered by observations of past access patterns. Such actions can affect performance but not the correctness of the protocol. Even the liveness of the protocol is not affected by voluntary rules, as long as some fairly weak fairness assumptions can be made. The system can switch from one microprotocol to another in a seamless and efficient manner, again based on some
observed program behavior or memory access patterns, thereby adding a further level of adaptivity.

Instructions in the processor either get executed by the processor right away, or are stalled because it needs to interact with memory. If an instruction gets stalled, a message is sent to the cache attached to that processor; when a response comes back to the processor, the stalled instruction can complete.

Cache can either respond to a processor request quickly, such as a read or write request on a location in cache, or slowly, such as when there is a cache miss. On a cache miss, the cache will send a request to memory for the data location. When a response returns from memory, the cache will extract the value from this response and deposit that value into its cache in the clean state, and then send a message to the processor to let it know that its stalled instruction can now complete.

When the memory receives a cache request or writeback message, either it immediately sends a response message or it sends messages to other processors' caches to change their state. When memory receives a purge acknowledgement message, it can change its internal state, without sending any messages out.

When processors receive a purge message from memory they change their state, and send a return message to memory. When memory receives the last purge ack message, it changes its state, and completes the writeback process with one final message to the originating party.

What we wish to synthesize is the sache controller and the memory controller. In Figure 3.1, these parts are highlighted, as the sections of the system that we wish to synthesize. For simplicity, we will synthesize the sache controller with the data of the sache attached; similarly we will implement the memory controller with the data of the memory attached. The synthesis of sache must deal with its internal state and the four ports: an input port from processor, an input port from memory, an output port to processor, and an output port to memory. To simplify synthesis, we want to assume zero buffering at the input ports, i.e.; there is no place to put aside a message if it is not processed. In a few situations in Shen's Cachet protocol, messages need to be put aside, so that they don't block other incoming messages. In such cases, we modify his protocol by sending a 'retry' message back to the source. We will discuss each such case in the context of specific protocols.
In the following sections, an explanation of the Cachet-Base protocol will be given, followed by explanations for Cachet-WriterPush and Cachet-Migratory. Some portions of the two latter microprotocols are similar to Cachet-Base, as described in section 3.2, and need no description. Only important differences are discussed in sections 3.3 and 3.4.

### 3.2 Description of Cachet-Base

The Cachet-Base cache coherence protocol is the simplest way to look at caching. It implements the CRF model without needing to store any state on the memory side; this microprotocol uses memory as a rendezvous point for data. A commit in this protocol pushes the dirty data from cache to main memory and a reconcile purges the clean data from the cache. This protocol is best used as a 'fall back' for when the implementation is unable or unwilling to store state on the memory side, or if the directory runs out of space at memory. This microprotocol is ideal when only necessary commits and reconciles are performed. The Cachet-Base Protocol is described in Figures 3.2, 3.3, and 3.4, on the following pages.

The two forms of rules in this thesis are Imperative and Directive. Imperative rules maintain the soundness and correctness of the cache coherence protocol. Directive rules maintain the liveness aspect of the cache coherence protocol.
Directive Rules for Sache Controller

The part of the protocol that discusses cache interaction with the processor is the set of Directive rules of Cachet-Base. As can be seen in Figure 3.2, most of the instructions that the processor issues to cache are immediately retired, so the next instruction can be issued. The way that an instruction is retired is that either a value or an acknowledgement is sent to the processor. For those instructions that cannot be retired immediately, a message is sent to memory so that when a response comes back, it can retire.

Most LoadL, StoreL and Commit instructions are immediately retired. The exceptions are when we have a LoadL or StoreL on location not in cache, or a Commit on a Dirty location in cache. All Reconciles are immediately retired, although the Clean Reconcile alters the cache state.

If the instruction is a LoadL on a location in cache, it is retired immediately. If the instruction is a StoreL on a location in cache, it changes the data of the location, as well as making the status of that location to be dirty (whether it started out dirty or clean), and is then immediately retired. If it is a LoadL or StoreL instruction on a location that is not in cache, a cache request message is sent to memory. When the response returns, then the location will be in cache as clean. If the instruction was a LoadL, it retires at that time, and if it was a StoreL, it will change the data and make the status of the cell dirty before it retires.

If we have a Reconcile instruction, it will immediately retire, as stated above, and if the state of the location is clean, then it is invalidated in the cache.

If we get a Commit instruction on a location that is dirty, a writeback request is sent to memory. When the writeback acknowledgement returns, the cache location has its status changed to clean, and the dirty commit can retire.

Imperative Rules for Sache Controller

The Imperative rules are the ones that are necessary to ensure the correctness of a protocol. The two rules in Figure 3.3 state the Imperative rules of Cachet-Base. They describe how the sache will react to a message from memory. The result of a cache message, coming to a site from memory, is to place the value that comes with the message into the site's sache, in the clean state. The result of a writeback acknowledgement is to change the status of that cell from writeback pending to clean.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cstate</th>
<th>MSG-&gt;Mem, Action</th>
<th>Next Cstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS1 &lt;LoadL,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>BS2 &lt;LoadL,a&gt;</td>
<td>Cell(a,v,Dirt)</td>
<td>retire</td>
<td>Cell(a,v,Dirt)</td>
</tr>
<tr>
<td>BS3 &lt;LoadL,a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>BS4 &lt;LoadL,a&gt;</td>
<td>Cell(a,-, CachePending)</td>
<td>stall</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td>BS5 &lt;LoadL,a&gt;</td>
<td>a not in cache, cache not full</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td>BS6 &lt;LoadL,a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,-, Clean)</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td>BS7 &lt;LoadL,a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,v,Dirty)</td>
<td>&lt;Wb(v),a1&gt; &amp; stall</td>
<td>Cell(a1,v,WbPending)</td>
</tr>
<tr>
<td>BS8 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Dirty)</td>
</tr>
<tr>
<td>BS9 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,v,Dirt)</td>
<td>retire</td>
<td>Cell(a,v,Dirt)</td>
</tr>
<tr>
<td>BS10 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>BS11 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,-, CachePending)</td>
<td>stall</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td>BS12 &lt;StoreL(v),a&gt;</td>
<td>a not in cache, cache not full</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td>BS13 &lt;StoreL(v),a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,v,Clean)</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td>BS14 &lt;StoreL(v),a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,v,Dirty)</td>
<td>&lt;Wb(v),a1&gt; &amp; stall</td>
<td>Cell(a1,v,WbPending)</td>
</tr>
<tr>
<td>BS15 &lt;Commit,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>BS16 &lt;Commit,a&gt;</td>
<td>Cell(a,v,Dirt)</td>
<td>stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>BS17 &lt;Commit,a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>BS18 &lt;Commit,a&gt;</td>
<td>Cell(a,-, CachePending)</td>
<td>stall</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td>BS19 &lt;Commit,a&gt;</td>
<td>a not in cache</td>
<td>retire</td>
<td>a not in cache</td>
</tr>
<tr>
<td>BS20 &lt;Reconcile,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell&lt;--Invalid</td>
</tr>
<tr>
<td>BS21 &lt;Reconcile,a&gt;</td>
<td>Cell(a,v,Dirt)</td>
<td>retire</td>
<td>Cell(a,v,Dirt)</td>
</tr>
<tr>
<td>BS22 &lt;Reconcile,a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>BS23 &lt;Reconcile,a&gt;</td>
<td>Cell(a,-, CachePending)</td>
<td>stall</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td>BS24 &lt;Reconcile,a&gt;</td>
<td>a not in cache</td>
<td>retire</td>
<td>a not in cache</td>
</tr>
</tbody>
</table>

Figure 3.2 Directive Sache Controller Protocol of Cachet-Base

<table>
<thead>
<tr>
<th>Msg from Mem</th>
<th>Cstate</th>
<th>Action</th>
<th>Next Cstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS25 &lt;Cache(v),a&gt;</td>
<td>Cell(a,-, CachePending)</td>
<td>unstall</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>BS26 &lt;WbAck,a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>unstall</td>
<td>Cell(a,v,Clean)</td>
</tr>
</tbody>
</table>

Figure 3.3 Imperative Sache Controller Protocol of Cachet-Base
Memory Controller

The two rules governing the memory behavior in Cachet-Base are given in Figure 3.4. They specify how memory reacts to a message that comes in from a site. If the message is a cache request message, we pull the value of the address out of the message, and use it to find the cell in memory. We read the value out of the cell of memory, and send a message back to the requesting site with that value. If the message is a writeback request, then we read the address and value out of the message. We use that address to find the cell in memory to write to, and we replace the value stored in that cell with the value in the message. Lastly, we send a message back to the site to notify it that the writeback has completed.

3.3 Description of Cachet-WriterPush

Read operations are usually more common than write operations, so, in Cachet-WriterPush, we wish to allow a Reconcile instruction to complete even when the address is cached in the clean state, without having to purge it from cache, which is what we had to do in Cachet-Base. Thus, subsequent reads on that location will not cause a cache-miss. However, a consequence of this faster processing of reads is an increased cost for writes. Since Cachet-WriterPush requires that all clean cells contain exactly the same data as main memory, when a Commit instruction is performed on a dirty copy of data, all the clean copies of the address must be purged from all other caches before the commit can complete. Therefore, committing an address cached in the Dirty state can be a lengthy process.

There are four new types of messages in this protocol, which did not show up in Cachet-Base. The purge request message, asks a site to get rid of a specified location in its cache. A purge acknowledgement message is what the site sends back to memory when it gets a purge request and removes the location from its cache. A writeback flush message tells a site that the writeback it requested was
preempted by another writeback, so the location in question must be removed from cache. The fourth new type of message, the retry message, did not show up in Xiaowei Shen’s protocols; it is one that this thesis introduces. This Retry message shows up in the WriterPush protocol, due to the possibility that data can be in a transient state. If a cache request message arrives at memory while in a transient state, a Retry message is sent to the site, and when the Retry message is received, the processor waits a time and reissues its cache request. The waiting for a time, however, is accomplished by having the Retry message invalidate the sache cell in question. This causes the initial rule, either a load or store that found that location missing from sache, to fire again and send off a cache request message.

The Cachet-WriterPush Protocol is shown below in Figures 3.5, 3.6, and 3.7. This protocol is ideal for programs with excessive use of Reconcile and LoadL operations. For example, if a processor reads a memory location many times before another processor modifies the location, this protocol behaves extremely favorably. A cache cell never needs to be purged from the cache unless the cache becomes full or another processor modifies the memory location.

**Sache Controller**

As with the directive rules for Cachet-Base microprotocol, most of the instructions issued by the site's processor are immediately retired for the Cachet-WriterPush microprotocol, as well. If an instruction cannot be immediately retired, a message is sent to memory so that when a response comes back, the instruction can be retired. The difference between Cachet-Base and Cachet-WriterPush in the directive rules is only on the Clean Reconcile rule: in Cachet-Base we invalidated that location, in effect ejecting the location from cache, while in Cachet-WriterPush we keep the location in sache in a clean state. We can see this in Figure 3.5.

When a site gets a Retry message from memory, the sache should have that location in a cache pending state. The sache invalidates the cell, so that the rule that sends a cache request must fire off again.

A processor that sends a writeback message to memory can get one of two responses: a writeback acknowledgement or a writeback flush. A writeback flush can be received if two writebacks get to memory at the same time; one of them gets to complete and gets a write back acknowledgement and the other must be told to flush its cache copy. The writeback flush is the message that is sent to the unlucky site, and the result of a writeback flush is invalidation of that location in sache.
If a site receives a purge request message regarding a location that is stored in cache in a clean state, then the location in cache is invalidated and a purge acknowledgement is sent back to memory.

If a purge request regarding a location stored in cache in the dirty state is received, then the status of the cell is changed to writeback pending, while a writeback message is sent to memory with the dirty value.

If a site gets a purge request message regarding a location in cache that is either in the writeback pending state or missing from the cache, then nothing is done other than removing the message.

**Memory Controller**

In Cachet-WriterPush, there are three types of requests that can arrive at memory: a cache request, a writeback, and a purge acknowledgement. Cache requests and writebacks have been explained in the discussion of Cachet-Base, but they are different in Cachet-WriterPush, so first an explanation of cache request messages will come, then an explanation for writeback messages, and lastly an explanation for purge acknowledgement messages.

A cache request requires the memory to add the sending site to the list of those who have an outstanding copy of that location, then send a message to the site with the data in that location.

A writeback message from a processor means that a processor wishes to write back data to the memory's store of the data. There are three cases, based on the state of that location in memory that will cause this message to be handled differently. The first case is when the memory cell of that location is currently stable, and only one site is in the list. In this case, the writeback has come from that one site in the list, and so that site had exclusive ownership of the data, so we can write the value that he wishes to be written and send a writeback acknowledgement back to him.

The second case is when memory is stable but has more than one site in the list. Before the writeback can complete, purge messages must be sent to all sites holding a copy of the location. Since the memory has to wait for these messages to be sent and then return, the memory cell must go into a transient state. This transient state's purpose is to block incoming cache requests against that location, since they cannot be handled while a writeback is occurring. The state will be restored to stable after the writeback completes, which will happen when all the purge messages are responded to. The transient state remembers who requested
This writeback, as well as what value he wished to write, and it keeps a list of who
still has outstanding copies of the data. This list will shrink as purge
acknowledgements return.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cstate</th>
<th>MSGs--&gt;Mem, Action</th>
<th>Next Cstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS1 &lt;LoadL,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,y,Clean)</td>
</tr>
<tr>
<td>WS2 &lt;LoadL,a&gt;</td>
<td>Cell(a,y,Dirty)</td>
<td>retire</td>
<td>Cell(a,y,Dirty)</td>
</tr>
<tr>
<td>WS3 &lt;LoadL,a&gt;</td>
<td>Cell(a,y,WbPending)</td>
<td>stall</td>
<td>Cell(a,y,WbPending)</td>
</tr>
<tr>
<td>WS4 &lt;LoadL,a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>WS5 &lt;LoadL,a&gt;</td>
<td>a not in cache, cache not full</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>WS6 &lt;LoadL,a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,-,Clean)</td>
<td>&lt;Purged,a1&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>WS7 &lt;LoadL,a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,v,Dirty)</td>
<td>&lt;Wb(v),a1&gt; &amp; stall</td>
<td>Cell(a1,v,WbPending)</td>
</tr>
<tr>
<td>WS8 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Dirty)</td>
</tr>
<tr>
<td>WS9 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,y,Dirty)</td>
<td>retire</td>
<td>Cell(a,y,Dirty)</td>
</tr>
<tr>
<td>WS10 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,y,WbPending)</td>
<td>stall</td>
<td>Cell(a,y,WbPending)</td>
</tr>
<tr>
<td>WS11 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>WS12 &lt;StoreL(v),a&gt;</td>
<td>a not in cache, cache not full</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>WS13 &lt;StoreL(v),a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,v,Clean)</td>
<td>&lt;Purged,a1&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>WS14 &lt;StoreL(v),a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,v,Dirty)</td>
<td>&lt;Wb(v),a1&gt; &amp; stall</td>
<td>Cell(a1,v,WbPending)</td>
</tr>
<tr>
<td>WS15 &lt;Commit,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>WS16 &lt;Commit,a&gt;</td>
<td>Cell(a,y,Dirty)</td>
<td>&lt;Wb(v),a&gt;, stall</td>
<td>Cell(a,y,WbPending)</td>
</tr>
<tr>
<td>WS17 &lt;Commit,a&gt;</td>
<td>Cell(a,y,WbPending)</td>
<td>stall</td>
<td>Cell(a,y,WbPending)</td>
</tr>
<tr>
<td>WS18 &lt;Commit,a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>WS19 &lt;Commit,a&gt;</td>
<td>a not in cache</td>
<td>retire</td>
<td>a not in cache</td>
</tr>
<tr>
<td>WS20 &lt;Reconcile,a&gt;</td>
<td>Cell(a,y,Clean)</td>
<td>retire</td>
<td>Cell(a,y,Clean)</td>
</tr>
<tr>
<td>WS21 &lt;Reconcile,a&gt;</td>
<td>Cell(a,y,Dirty)</td>
<td>retire</td>
<td>Cell(a,y,Dirty)</td>
</tr>
<tr>
<td>WS22 &lt;Reconcile,a&gt;</td>
<td>Cell(a,y,WbPending)</td>
<td>stall</td>
<td>Cell(a,y,WbPending)</td>
</tr>
<tr>
<td>WS23 &lt;Reconcile,a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>WS24 &lt;Reconcile,a&gt;</td>
<td>a not in cache</td>
<td>retire</td>
<td>a not in cache</td>
</tr>
</tbody>
</table>

Figure 3.5 Directive Sache Controller Protocol of Cachet-WriterPush
The third case is when a writeback request gets to memory while in the transient state. In this case, there are two competing writebacks that wish to write data to memory. Obviously only one value will be written to memory, and either value that is written is correct from sequential consistency view of memory. In the model for this work, the later writeback always loses this contention, so a writeback flush message is sent to this latter site, after being removed from the list of sites. Additionally, if this writeback request is coming from the last site on the list of sites that hold this data, then both a writeback flush and a writeback acknowledgement must be sent out.

A purged message indicates that the sending site has purged a location from sache, so it should be removed from the list of sites that have that location. If the memory state is stable, no messages are sent after removing the site from the
list. If the memory's state is transient, there are two cases. In the case where there are still outstanding copies, the rule only removes the site from the list. If this is the last site in the list, however, then the purging process is complete, and the writeback that initiated the purges can now complete. So, the value that was sent in the original writeback message is made to be the value of the cell and the state of the cell becomes stable. The only site in the new stable list is the site that sent the writeback. Lastly, a message is sent to this site, letting it know the writeback completed successfully.

### 3.4 Description of Cachet-Migratory

If an address is accessed repeatedly by only one processor, for a reasonable time period, then that site should get exclusive access to that location, so that all instructions on the address become local instructions, that complete cheaply and quickly. The protocol is reminiscent of the Exclusive state in the conventional MESI protocol, since an address is cached in at most one cache at any time. Thus, both Reconcile and Commit instructions on a location in cache can complete right away. The price we pay for this nice feature shows up when we get a cache miss in another cache. At that time, we must migrate the exclusive ownership to the other cache, which is expensive. The Cachet-Migratory Protocol is described below, in Figures 3.8, 3.9 and 3.10.

**Sache Controller**

The directive rules for Cachet-Migratory are, like for Cachet-WriterPush, nearly a carbon copy of those for Cachet-Base. Just as Cachet-WriterPush, the rule for Clean Reconcile is such that we keep the location in sache in a clean state. However, a change from both of the other protocols is that we do nothing when we get a Dirty Commit. We retire the commit without sending any messages to memory or changing the state of the sache cell.

There are three types of messages the processor can get from memory in Cachet-Migratory. Cache messages and purge-request messages are entirely similar to those for Cachet-WriterPush, so no explanation is given. However, when a site gets a Retry message from memory, the sache should have that location in a cache pending state. The retry message causes the sache to invalidate the cell, and at some point, the rule that initially caused the cache request to be sent will fire.
again. We could have instead re-sent the cache request to memory immediately. However, then we would be utilizing the network as a buffer, which is not good.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cstate</th>
<th>MSGs-&gt;Mem, Action</th>
<th>Next Cstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS1 &lt;LoadL,a&gt;</td>
<td>Cell(a,y,Clean)</td>
<td>retire</td>
<td>Cell(a,y,Clean)</td>
</tr>
<tr>
<td>MS2 &lt;LoadL,a&gt;</td>
<td>Cell(a,y,Dirty)</td>
<td>retire</td>
<td>Cell(a,y,Dirty)</td>
</tr>
<tr>
<td>MS3 &lt;LoadL,a&gt;</td>
<td>Cell(a,_,CachePending)</td>
<td>stall</td>
<td>Cell(a,_,CachePending)</td>
</tr>
<tr>
<td>MS4 &lt;LoadL,a&gt;</td>
<td>a not in cache, cache not full</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,_,CachePending)</td>
</tr>
<tr>
<td>MS5 &lt;LoadL,a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,_,Clean)</td>
<td>&lt;Purged,a&gt; &amp; stall</td>
<td>Cell(a,_,CachePending)</td>
</tr>
<tr>
<td>MS6 &lt;LoadL,a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,y,Dirty)</td>
<td>&lt;Wb(v),a&gt; &amp; stall</td>
<td>Cell(a,_,CachePending)</td>
</tr>
<tr>
<td>MS7 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,y,Clean)</td>
<td>retire</td>
<td>Cell(a,y,Dirty)</td>
</tr>
<tr>
<td>MS8 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,y,Dirty)</td>
<td>retire</td>
<td>Cell(a,y,Dirty)</td>
</tr>
<tr>
<td>MS9 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,_,CachePending)</td>
<td>stall</td>
<td>Cell(a,_,CachePending)</td>
</tr>
<tr>
<td>MS10 &lt;StoreL(v),a&gt;</td>
<td>a not in cache, cache not full</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,_,CachePending)</td>
</tr>
<tr>
<td>MS11 &lt;StoreL(v),a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,y,Clean)</td>
<td>&lt;Purged,a&gt; &amp; stall</td>
<td>Cell(a,_,CachePending)</td>
</tr>
<tr>
<td>MS12 &lt;StoreL(v),a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,y,Dirty)</td>
<td>&lt;Wb(v),a&gt; &amp; stall</td>
<td>Cell(a,_,CachePending)</td>
</tr>
<tr>
<td>MS13 &lt;Commit,a&gt;</td>
<td>Cell(a,y,Clean)</td>
<td>retire</td>
<td>Cell(a,y,Clean)</td>
</tr>
<tr>
<td>MS14 &lt;Commit,a&gt;</td>
<td>Cell(a,y,Dirty)</td>
<td>retire</td>
<td>Cell(a,y,Dirty)</td>
</tr>
<tr>
<td>MS15 &lt;Commit,a&gt;</td>
<td>Cell(a,_,CachePending)</td>
<td>stall</td>
<td>Cell(a,_,CachePending)</td>
</tr>
<tr>
<td>MS16 &lt;Commit,a&gt;</td>
<td>a not in cache</td>
<td>retire</td>
<td>a not in cache</td>
</tr>
<tr>
<td>MS17 &lt;Reconcile,a&gt;</td>
<td>Cell(a,y,Clean)</td>
<td>retire</td>
<td>Cell(a,y,Clean)</td>
</tr>
<tr>
<td>MS18 &lt;Reconcile,a&gt;</td>
<td>Cell(a,y,Dirty)</td>
<td>retire</td>
<td>Cell(a,y,Dirty)</td>
</tr>
<tr>
<td>MS19 &lt;Reconcile,a&gt;</td>
<td>Cell(a,_,CachePending)</td>
<td>stall</td>
<td>Cell(a,_,CachePending)</td>
</tr>
<tr>
<td>MS20 &lt;Reconcile,a&gt;</td>
<td>a not in cache</td>
<td>retire</td>
<td>a not in cache</td>
</tr>
</tbody>
</table>

Figure 3.8 Directive Sache Controller Protocol of Cachet-Migratory

<table>
<thead>
<tr>
<th>Msg from Mem</th>
<th>Cstate</th>
<th>Action</th>
<th>Next Cstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS21 &lt;Cache(v),a&gt;</td>
<td>Cell(a,_,CachePending)</td>
<td>unstall</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>MS22 &lt;Retry,a&gt;</td>
<td>Cell(a,_,CachePending)</td>
<td>unstall</td>
<td>Cell&lt;--Invalid</td>
</tr>
<tr>
<td>MS23 &lt;PurgeReq,a&gt;</td>
<td>Cell(a,_,Clean)</td>
<td>&lt;Purged,a&gt;</td>
<td>Cell&lt;--Invalid</td>
</tr>
<tr>
<td>MS24 &lt;PurgeReq,a&gt;</td>
<td>Cell(a,y,Dirty)</td>
<td>&lt;Wb(v),a&gt;</td>
<td>Cell&lt;--Invalid</td>
</tr>
<tr>
<td>MS25 &lt;PurgeReq,a&gt;</td>
<td>a not in cache</td>
<td>none</td>
<td>a not in cache</td>
</tr>
</tbody>
</table>

Figure 3.9 Imperative Sache Controller Protocol of Cachet-Migratory
There are three types of messages that memory can receive in Cachet-Migratory, and they are cache request messages, writeback messages, and purge acknowledgement messages.

A cache request coming to memory can either find the location being held by nobody, by another site, or transient. When a cache request message comes to memory and no one currently has a copy of that location, then the tag in memory gives exclusive access to the requesting site, and sends a cache message to that site with the value in memory. When that location is already being held by another site, then the memory cell goes to the MigTrans state, and sends a purge message to the site that holds the location. The MigTrans state records both the id of the site requesting the location and the id of the site that holds the location. When a cache request message comes to memory and the memory cell is in a transient state, then a retry message is sent back to the site.

If memory receives a writeback request for a location that is in the MigTrans state, then that means that this writeback is coming as the result of a purge message sent by memory. Memory changes the value in the cell to the value that was carried back by the writeback message. The state in memory becomes exclusive for the site that requested the migration of ownership, and a cache message is sent to that site, with the newest value for that location.

If memory receives a writeback request for a location that is in the Exclusive state, that means this writeback is not the result of a request by another site to get access to the data. This could happen if the site containing the exclusive copy gets its cache to a saturated point and has to decide to throw out something—the location that is thrown out gets written back to memory. The memory writes the value from the writeback message into the cell, and tags it with Noone, indicating that no one has a copy of this data.
If memory receives a purged message for a location that is in the Exclusive state, it behaves similarly to when it receives a writeback in Exclusive state (and can occur in the same fashion: cache overflow). There is no data to write back, so the state of the cell is just changed from Exclusive to Noone.

Lastly, if memory receives a purged message for a location that is in the MigTrans state, it is the result of a purge request, and it is handled similarly to when a writeback comes back to a MigTrans state, a few paragraphs above. The status of the memory cell for the location is changed to exclusive for the site that requested migration of ownership, and a message is sent to that site with the value in that location.
Chapter 4
Synthesis of Cachet Microprotocols

The first step of this thesis was to take Shen's TRS description of Cachet, and make an instantiation of his protocol. The second step is to specify and define the interfaces for the sache controller and for the memory controller, in the environment of the more implementable protocol. The third step is to extract TRS rules from the implementable protocol rules that will use the specified interfaces to implement the sache controller and memory controller. We will then take these rules and send them through the TRAC compiler, to get RTL code. The next step would be to run the RTL code on a Real Circuit Simulator. This Simulator will tell us how well the RTL code would perform, if we built actual working hardware and ran the Cachet protocol on that hardware. This Chapter discusses the second and third steps.

4.1 Signaling Convention and Rule Example

We wish to synthesize the memory, with its one input and one output, and the sache, with its two inputs and two outputs. The signaling convention that has been implemented in this work is to use a full asynchronous handshake on input and output ports.

To explain a full asynchronous handshake, we illustrate the passing of a message from Module A to Module B, in Figure 4.1, below. The tuple that would relate to the message diagrammed is <pcm, pcx, pcy> [pcm means Processor to Cache Message]. The transmission line between A and B, is such that it is an output port on Module A and an input port on Module B. The bits that handle the handshaking are labeled pcx and pcy. They have a line of implied communication between the modules, such that if Module A changes pcx, then Module B can see the change, in what we call the shadow register of pcx. It keeps the 'last' value of pcx in pcy. This allows Module B to do a comparison of bits in pcy and pcx to see whether or not they are equal, without going outside the module. Module A can
only change the real register pcx and not the shadow register pcy. By convention, messages can be sent from Module A only when pcx is equal to pcy, and can be received by Module B only when the two registers are different. Thus, for Module A to send a message, we need to check if pcx equals pcy, and for Module B to receive a message, we need to check if pcx is not equal to pcy.

The input port of memory can have a message on it, and if the bits following it in the data structure differ, then this message has not yet been read. If this message is new, it can be read and the internal memory state can change, while the memory also sends out a message. When this is all done, the message has been processed and the bits are set to be equal again. The output port of memory is similar to the input port; it can have an output message put onto it, but only if the bits following it are the same. If they are the same, then the last message that was on the output port has been taken, so it is safe to send out a new message. If we did not check these bits, a new output message would overwrite a previous, and unread message. When a new output message is put out, one of the bits are toggled to make them differ again. So, messages being sent can only go out if the bits are the same, and if they are they are set to be different, and incoming messages can only be read if the bits differ, and when the message is read the bits are made to be the same.

This type of comparison shows up in the rules given later again and again. For example, the rule shown in Figure 4.2 has ((cpx == cpy) && (pcx != pcy)) as its predicate. This line checks to see that pcx and pcy are different, so the pcm
message in the tuple \(<\text{pcm},\text{pcx},\text{pcy}>\) is a new, unread message from processor to cache. It also checks that \(\text{cpx}\) is the same as \(\text{cpy}\); which would mean that the last \(\text{cpm}\) message on the cache to processor output port has been processed, so we can send a new output message on this port. Since Clean LoadL requires both reading of an input on PC input port and sending of output on CP output port, both of these checks (\(\text{cpx}\) equal to \(\text{cpy}\) and \(\text{pcx}\) not equal to \(\text{pcy}\)) must be done for the rule to fire.

\[
\text{Rule "Base Clean or Dirty LoadL"}
\]

\[
\text{Sache(Pcs(Pcm(t,LoadL(a),\text{pcx},\text{pcy}),Cps(-,\text{cpx},\text{cpy}),mcs,cms,\text{data})}
\]

\[
\text{if } ((\text{pcx} \neq \text{pcy}) \&\& (\text{cpx} == \text{cpy})) \&\& (\text{DorC(s)}) )
\]

\[
\text{where Cell(a,v,s) = data[hash(a)]}
\]

\[
===>\text{Sache(Pcs(-,\text{pcx},\text{pcx}),Cps(Cpm(t,Value()),not(\text{cpx}),\text{cpy}),mcs,cms,\text{data})}
\]

Figure 4.2 Rule Example

The figure above illustrates a rule and an explanation is given afterwards of how various components will generally be used in this thesis. First we have the label for the rule, in this case the rule is called Base Clean or Dirty LoadL. This rule applies to the Cachet-Base microprotocol, and will fire when incoming message from processor is a LoadL, when the address that is being referenced exists in the cache in a clean or dirty state. The left-hand-side (LHS) of the rule consists of the initial state before the rule fires. It includes some variable definitions and qualifiers that need to be satisfied for the rule to fire, as well. The right-hand-side (RHS) of the rule is the next state, which replaces the state that was given in the LHS.

For this above rule, a pattern match must be satisfied to match the sache type given in the LHS of the rule. The rest of the LHS is pattern matching and variable assignment, and this is the part of the rule that starts with the key word 'where'. For this rule, if it fires, simply sends a message back to the processor with the value stored in cache.
What we wish to do is to synthesize the memory controller and the sache controller components of a multiprocessor system. Such a system has a system memory, and a number of sites, which are processing programs that are set up at those sites. In this model, there are a specific number of sites because TRSpec does not handle arrays of complex data types. A diagram of the system, which shows the components we wish to synthesize, is shown above. Each site has a semantic cache, which has interfaces to the processor and memory. We wish to synthesize the memory, with its one input and one output, and the sache, with its two inputs and two outputs. In Figure 4.3, we illustrate how we abstract away all of the issues of buffering, bussing, network message transfer, and message protocols, to simply call the middle area the Network module. We expect it to ‘do the right thing’. That is to say, we will send out messages on our output port and make the assumption that it will arrive at the destination after some finite delay, without being lost. Given these assumptions, we do not need to concern ourselves with the network anymore, and while it could also be synthesized, that is not the focus of this work. The focus of this work is the synthesis of the sache controller and the
synthesis of the memory controller. We need to have the network, as an abstract, defined so that we know what inputs and outputs each of these components will have. This will assist us in creating the state definitions (which implicitly include the interface definitions) for the sache and memory controllers.

### 4.2 Memory Controller State Definition

In Figure 4.4, the state definition of the memory module is stated. Embedded in state definition are the implicit assumptions for the input and output interfaces, so in a state definition is also our interface definition.

The memory section of a system has an array of MemCells, indexed by an address (type ADDR is a 16-bit number), and a state tag, for when the memory goes into a transitory state. When multiple messages must be produced, due to a single input message, memory may have to go into a transitory state, since only one message at a time can be put on the output port of the memory component. The possibilities for the state tag are Nil, which is the case in regular operation, WbAcking, which occurs when we need to send a writeback flush and a writeback ack message both at once, and the states Purgings and Purging, which indicate that there are still purging messages, or a purging message, respectively, to be sent out.

Each MemCell has an implicit address (the index by which you reach the MemCell), a value (type VAL is a 16-bit number, as well), and a tag associated with it. The enumeration for the tag is slightly more complex: the cell can be in one of 5 states, and two of these are used by Cachet-WriterPush, while the other three are used by Cachet-Migratory. Cachet-Base does not access or touch this tag, because in Cachet-Base, no state is needed at memory.

The three tags used by Cachet-Migratory are Noone, indicating that no one has a copy of this data, Exclusive, which means that one (and only one) of the processors has an exclusive copy of that location right now, and MigTrans, which means that that location is in a transient state right now. Exclusive and MigTrans have as part of the tag the actual number of the processor who has this data out now. The type NUMSITE that was used for this is a 2-bit number, which represents the numbers 0 to 3, which are the numbers used for the four sites.
The two tag types used by Cachet-Writer push are Stable and Transient. The best way to implement the Stable state would be with a content-addressable list or sequence. However, due to limitations of TRAC, a simpler data structure was required. A four-way on-off switch type of data structure was used, in which there is a semantic yes or no (1 or 0) in each of the positions in Stable(x,x,x,x) which correspond to sites 0 through 3, respectively. If any of these is a 1, this corresponds to having a clean copy of data in the site that is represented by its position in the list of sites (0 to 3). Transient is an extension of Stable, in which we also store the number of the processor requesting a writeback, and the data which the processor wishes to write onto the current data in memory. These are necessary to be stored at the beginning of transmuting a stable state to transient (which occurs when a writeback request comes to memory), so that when all purges return to memory, it is known where to send the writeback acknowledgement and what data to write in the cell at memory.

An input message has a tag for which kind of message it is, as well as the site sending the message and the memory address that is being referenced. The tag indicates whether we have a writeback, a cache request, or a purge acknowledgement coming to memory from a site.
An output message has a tag for which kind of message it is, as well as the destination site, and the address being referenced by the message. The tag indicates whether we have a write back acknowledgement, cache request acknowledgement, write back flush message or a purge request. The acknowledgement messages, are, of course, self-explanatory. When a writeback request for a location comes to memory, then purge requests are sent to all processors holding clean copies of the data. If two writebacks arrive at memory at nearly the same time, one of the writebacks will fail; and the fail message that is sent is the writeback flush. This writeback flush message should be implemented as a purge message instead, which would reduce the number of types. However, since the description as currently stated works, and doing this may cause an unforeseen bug, this change was not be made.

4.3 Sache Controller State Definition

In Figure 4.5, the state definition needed to describe a sache is stated. A sache is just a semantic cache, and a sache module has two input ports, one from processor (pc_) and one from memory (mc_), two output ports, one to processor (cp_) and one to memory (cm_), and of course the storage for the cells in the cache. Each cell has the address of memory that is being stored here, the value at that address, and the current status of the data, as represented by the CellTag. The CellTag can be the standard clean and dirty states, as well as states that indicate something is being waited upon, which are cachepending and wbpending.

The mc input is for messages incoming from memory to this site, and it is very similar to messages in the outgoing buffer at the memory side. These messages contain a tag, labeling which type of message it is, as well as two synchronization bits that utilize the signaling convention explained in Section 4.1. The tag on the message is similar to tags on the outgoing buffer on memory, and the possibilities are the same: writeback acknowledgement, cache request acknowledgement, writeback flush or a purge request. The only difference is that there is no destination processor. Since it is an input port on a specific site's sache, it is that site that is the destination.
<table>
<thead>
<tr>
<th>Type</th>
<th>SACHE</th>
<th>=</th>
<th>Sache(PCS, CPS, MCS, CMS, DATA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>DATA</td>
<td>=</td>
<td>Array [ADDR]CELL</td>
</tr>
<tr>
<td>Type</td>
<td>CELL</td>
<td>=</td>
<td>Cell(ADDR, VAL, CELLTAG)</td>
</tr>
<tr>
<td>Type</td>
<td>CELLLTAG</td>
<td>=</td>
<td>Invalid()</td>
</tr>
<tr>
<td>Type</td>
<td>PCS</td>
<td>=</td>
<td>Pcs(PCMYNYN)</td>
</tr>
<tr>
<td>Type</td>
<td>PCM</td>
<td>=</td>
<td>Pcm(PROCTAG, INST, ADDR)</td>
</tr>
<tr>
<td>Type</td>
<td>INST</td>
<td>=</td>
<td>LoadL()</td>
</tr>
<tr>
<td>Type</td>
<td>PROCTAG</td>
<td>=</td>
<td>Bit[16]</td>
</tr>
<tr>
<td>Type</td>
<td>CPS</td>
<td>=</td>
<td>Cps(CPM,YN,YN)</td>
</tr>
<tr>
<td>Type</td>
<td>CPM</td>
<td>=</td>
<td>Cpm(PROCTAG, VALACK)</td>
</tr>
<tr>
<td>Type</td>
<td>VALACK</td>
<td>=</td>
<td>Value(VAL)</td>
</tr>
<tr>
<td>Type</td>
<td>MCS</td>
<td>=</td>
<td>Mcs(MCM,YN,YN)</td>
</tr>
<tr>
<td>Type</td>
<td>MCM</td>
<td>=</td>
<td>Mcm(MCTAG, ADDR)</td>
</tr>
<tr>
<td>Type</td>
<td>MCTAG</td>
<td>=</td>
<td>WbAck()</td>
</tr>
<tr>
<td>Type</td>
<td>CMS</td>
<td>=</td>
<td>Cms(CMM,YN,YN)</td>
</tr>
<tr>
<td>Type</td>
<td>CMM</td>
<td>=</td>
<td>Cmm(CMTAG, ADDR)</td>
</tr>
<tr>
<td>Type</td>
<td>CMTAG</td>
<td>=</td>
<td>Wb(VAL)</td>
</tr>
</tbody>
</table>

Figure 4.5 Semantic Configuration of a Sache

The cm output is for messages outgoing from the sache to the memory, and it is very similar to the messages in the incoming buffer at memory. A cm message has a tag and an address. There is no source site, since this is an output port on a specific site's sache, it is that site that is the source site. The synchronization bits cmx and cmy implement the signaling convention.

The pc input is for instructions that the processor wished to be done. The messages that it sends are of the four basic type of instruction, namely LoadL, StoreL, Commit and Reconcile. All of these come with a reference address that the message refers to; this address is the address of the data in main memory that the processor wishes to access.

The cp output is for messages going from cache to processor. The two types of messages cache sends back to processor are values, which apply to LoadL instructions, and acknowledgements, which apply to the other three instructions.
4.4 Cachet-Base

Next, the rules implementing the sache and memory interfaces for the Cachet-Base protocol will be given.

4.4.1 Sache Controller of Cachet-Base

These rules correspond on a 1-to-1, or a many-to-1, basis with the specification given in the protocol, displayed in Figure 3.2. The cases where it is a many-to-1 correspondence are when we can collapse multiple cases easily into a single rule.

**Rule "BS1 + BS2 Base Clean or Dirty LoadL"**

\[
\begin{align*}
\text{Sache}(\text{Pcs}(\text{Pcm}(t, \text{LoadLO}, a), \text{pcx}, \text{pcy}), \text{Cps}(\cdot, \text{cpx}, \text{cpy}), mcs, cms, data) \\
\text{if } ((\text{pcx} =! \text{pcy}) \& \& (\text{cpx} =! \text{cpy}) \& \& (\text{DorC}(s))) \\
\text{where Cell(a,v,s) = data[hash(a)]} \\
\Rightarrow \text{Sache}(\text{Pcs}(-, \text{pcx}, \text{pcx}), \text{Cps}(\text{Cpm}(t, \text{Value}(v)), \not(\text{cpx}), \not(\text{cpy}), mcs, cms, data)
\end{align*}
\]

**Rule "BS5 Base Invalid LoadL"**

\[
\begin{align*}
\text{Sache}(\text{Pcs}(\text{Pcm}(t, \text{LoadLO}, a), \text{pcx}, \text{pcy}), \text{cps}, mcs, Cms(-, \text{cmx}, \text{cmx}, \text{data})) \\
\text{if } ((\text{pcx} =! \text{pcy}) \& \& (\text{cmx} =! \text{cmx}) ) \\
\text{where InvalidO} = \text{data[hash(a)]} \\
\Rightarrow \text{Sache}(\text{Pcs}(-, \text{pcx}, \text{pcy}), \text{cps}, mcs, Cms(\text{Cmm}(\text{CacheReqO}, a), \not(\text{cmx}), \text{cmx}), \\
\text{data}[\text{hash(a)}: = \text{Cell}(a, -, \text{CachePendingO})])
\end{align*}
\]

**Rule "BS6 Base Clean Collision LoadL"**

\[
\begin{align*}
\text{Sache}(\text{Pcs}(\text{Pcm}(t, \text{LoadLO}, a), \text{pcx}, \text{pcy}), \text{cps}, mcs, Cms(-, \text{cmx}, \text{cmx}, \text{data})) \\
\text{if } ((\text{pcx} =! \text{pcy}) \& \& (\text{cmx} =! \text{cmx}) \& \& (a1 =! a)) \\
\text{where Cell(a,1,-,CleanO) = data[hash(a)]} \\
\Rightarrow \text{Sache}(\text{Pcs}(-, \text{pcx}, \text{pcy}), \text{cps}, mcs, Cms(\text{Cmm}(\text{CacheReqO}, a), \not(\text{cmx}), \text{cmx}), \\
\text{data}[\text{hash(a)}: = \text{Cell}(a, 1, -, \text{CachePendingO})])
\end{align*}
\]

**Rule "BS7 Base Dirty Collisison LoadL"**

\[
\begin{align*}
\text{Sache}(\text{Pcs}(\text{Pcm}(t, \text{LoadLO}, a), \text{pcx}, \text{pcy}), \text{cps}, mcs, Cms(-, \text{cmx}, \text{cmx}, \text{data})) \\
\text{if } ((\text{pcx} =! \text{pcy}) \& \& (\text{cmx} =! \text{cmx}) \& \& (a1 =! a)) \\
\text{where Cell(a,1,-,DirtyO) = data[hash(a)]} \\
\Rightarrow \text{Sache}(\text{Pcs}(-, \text{pcx}, \text{pcy}), \text{cps}, mcs, Cms(\text{Cmm}(\text{Wb}(v), a1), \not(\text{cmx}), \text{cmx}), \\
\text{data}[\text{hash(a)}: = \text{Cell}(a, 1, v, \text{WbPendingO})])
\end{align*}
\]

**Rule "BS8 + BS9 Base Clean or Dirty StoreL"**

\[
\begin{align*}
\text{Sache}(\text{Pcs}(\text{Pcm}(t, \text{StoreL}(v), a), \text{pcx}, \text{pcy}), \text{Cps}(\cdot, \text{cpx}, \text{cpy}), mcs, cms, data) \\
\text{if } ((\text{pcx} =! \text{pcy}) \& \& (\text{cpx} =! \text{cpy}) \& \& (\text{DorC}(s))) \\
\text{where Cell(a,v,-,s) = data[hash(a)]} \\
\Rightarrow \text{Sache}(\text{Pcs}(-, \text{pcx}, \text{pcy}), \text{Cps}(\text{Cpm}(t, \text{AckO}), \not(\text{cpx}), \not(\text{cpy}), mcs, cms, \\
\text{data}[\text{hash(a)}: = \text{Cell}(a, v, \text{DirtyO})])
\end{align*}
\]
Rule "BS12 Base Invalid StoreL"
Sache(Pcs(Pcm(t, StoreL(u),a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
  if ((pcx != pcy) && (cmx == cmy))
    where Invalid() = data[hash(a)]
  ==> Sache(Pcs(-,pcx,pcy),cps,mcscms,Cms(Cmm(CacheReqO,a),not(cmcmx,cmy),
    data[hash(a):=Cell(a,-,CachePending())])

Rule "BS13 Base Clean Collision StoreL"
Sache(Pcs(Pcm(t, StoreL(u),a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
  if ((pcx != pcy) && (cmx == cmy)) && (al != a)
    where Invalid() = data[hash(a)]
  ==> Sache(Pcs(-,pcx,pcy),cps,mcscms,Cms(Cmm(Wb(v),a),not(cmcmx,cmy),
    data[hash(a):=Cell(a, v, WbPending())])

Rule "BS14 Base Dirty Collision StoreL"
Sache(Pcs(Pcm(t, StoreL(u),a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
  if ((pcx != pcy) && (cmx == cmy)) && (al != a)
    where Invalid() = data[hash(a)]
  ==> Sache(Pcs(-,pcx,pcy),cps,mcscms,Cms(Cmm(Wb(v),a),not(cmcmx,cmy),
    data[hash(a):=Cell(a1,v, WbPending())])

Rule "BS15 + BS19 Base Clean or CMiss Commit"
Sache(Pcs(Pcm(t, Commit(),a),pcx,pcy),Cps(-,cpx,cpy),mcscms,Cms(-,cmx,cmy),data)
  if ( ((pcx != pcy) && (cpx == cpy)) && (CorM(cs)) )
    where Invalid() = data[hash(a)]
  ==> Sache(Pcs(-,pcx,pcy),cps,mcscms,Cms(Cmm(Wb(v),a1),not(cmcmx,cmy),
    data[hash(a):=Cell(a1,v, WbPending())])

Rule "BS16 Base Dirty Commit"
Sache(Pcs(Pcm(t, Commit(),a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
  if ( ((pcx != pcy) && (cmx == cmy)) )
    where Invalid() = data[hash(a)]
  ==> Sache(Pcs(-,pcx,pcy),cps,mcscms,Cms(Cmm(Wb(v),a1),not(cmcmx,cmy),
    data[hash(a):=Cell(a1,v, WbPending())])

Rule "BS20 Base Clean Reconcile"
Sache(Pcs(Pcm(t, Reconcile(),a),pcx,pcy),Cps(-,cpx,cpy),mcscms,Cms(-,cmx,cmy),data)
  if ( ((pcx != pcy) && (cpx == cpy)) )
    where Invalid() = data[hash(a)]
  ==> Sache(Pcs(-,pcx,pcy),cps,mcscms,Cms(Cmm(Wb(v),a1),not(cmcmx,cmy),
    data[hash(a):=Invalid()])

Rule "BS21 + BS24 Base Dirty or CMiss Reconcile"
Sache(Pcs(Pcm(t, Reconcile(),a),pcx,pcy),Cps(-,cpx,cpy),mcscms,Cms(-,cmx,cmy),data)
  if ( ((pcx != pcy) && (cpx == cpy)) && (DorM(cs)) )
    where Invalid() = data[hash(a)]
  ==> Sache(Pcs(-,pcx,pcy),cps,mcscms,Cms(Cmm(Wb(v),a1),not(cmcmx,cmy),
    data[hash(a):=Invalid()])

Rule "BS25 Base Cache Data"
Sache(pcs,cps,Mcs(Mcm(Cache(v),a),mcx,mcy),mcscms,Cms(-,cmx,cmy),data)
  if (mcx != mcy)
    where CachePending() = data[hash(a)]
  ==> Sache(pcs,cps,Mcs(-,mcx,mcx),mcscms,Cms(-,cmx,cmy),data[hash(a):=Invalid()])
Rule "BS26 Base Write Back Done"
\[
\text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(\text{Mc} \text{m(WbAck)}(a), \text{mcx}, \text{mcy}), \text{cms}, \text{data})
\]
\[
\text{if } (\text{mcx} != \text{mcy})
\]
\[
\text{where Cell}(a, v, \text{WbPending})() = \text{data}[\text{hash}(a)]
\]
\[
\implies \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(-, \text{mcx}, \text{mcy}), \text{cms}, \text{data}[\text{hash}(a):=\text{Cell}(a, v, \text{Clean}(a))])
\]

4.4.2 Memory Controller of Cachet-Base

Rule "BM1 Cache Request"
\[
\text{SysMem(Ins(InM(CacheReq(), a, ws), inx, iny), Outs(-, outx, outy), memdata, Nil())}
\]
\[
\text{if } ((\text{inx} != \text{iny}) \&\& (\text{outx} == \text{outy}))
\]
\[
\text{where MemCell}(v, -) = \text{memdata}[a]
\]
\[
\implies \text{SysMem(Ins(-, \text{inx}, \text{inx}), Outs(OutM(Cache(v), a, ws), not(outx), outy),}
\]
\[
\text{memdata, Nil())}
\]

Rule "BM2 Write Back Request"
\[
\text{SysMem(Ins(InM(Wb(v), a, ws), inx, iny), Outs(-, outx, outy), memdata, Nil())}
\]
\[
\text{if } ((\text{inx} != \text{iny}) \&\& (\text{outx} == \text{outy}))
\]
\[
\implies \text{SysMem(Ins(-, \text{inx}, \text{inx}), Outs(OutM(WbAck(), a, ws), not(outx), outy),}
\]
\[
\text{memdata[a:=MemCell(v, -)]}, \text{Nil()})
\]

4.5 Cachet-WriterPush

This next section describes the rules implementing the sache and memory interfaces for the Cachet-WriterPush protocol. First, the rules of the Sache Controller will be explained, and following that will come the rules for the Memory Controller.

4.5.1 Sache Controller of Cachet-WriterPush

The only difference from Cachet-Base to Cachet-WriterPush in the processor generated rules is the Clean Reconcile. In Cachet-Base, we invalidate the location, but in Cachet-WriterPush, we keep the location in sache in the clean state.

Rule "WS15 + WS16 + WS19 Clean or Dirty or CMiss Reconcile"
\[
\text{Sache}(\text{Pcs(Pcm(t,Reconcile)}(a), \text{pcx}, \text{pcy}), \text{Cps(-,cpx,cpy)}, \text{cms}, \text{data})
\]
\[
\text{if } ((\text{pcx} != \text{pcy}) \&\& (\text{cpx} == \text{cpy})) \&\& (\text{CorDorM}(cs))
\]
\[
\text{where cs} = \text{data}[\text{hash}(a)]
\]
\[
\implies \text{Sache}(\text{Pcs(-,pcx,pcx}), \text{Cps(Cpm(t,Ack())}, \text{not(cpx),cpy}), \text{cms}, \text{data})
\]
Below are the rules to handle messages coming into the sache controller from memory. They follow directly from the Cachet-WriterPush protocol.

**Rule "WS25 CacheReq Response"**

\[ \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(	ext{Mcm}(\text{Cache}(v), a), mcx, mcy), \text{cms}, \text{data}) \]

if \((mcx \neq mcy)\)
where \(\text{Cell}(a, -, \text{CachePending}()) = \text{data}[\text{hash}(a)]\)

\[ \Rightarrow \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(-, mcx, mcx), \text{cms}, \text{data}[\text{hash}(a):=\text{Cell}(a, v, \text{Clean}())]) \]

**Rule "WS26 Retry"**

\[ \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(	ext{Mcm}(\text{Retry}(), a), mcx, mcy), \text{cms}, \text{data}) \]

if \(((mcx \neq mcy) \&\& (cmx == cmy))\)
where \(\text{Cell}(a, -, \text{CachePending}()) = \text{data}[\text{hash}(a)]\)

\[ \Rightarrow \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(-, mcx, mcx), \text{cms}, \text{data}[\text{hash}(a):=\text{Invalid}()]) \]

**Rule "WS27 Write Back Done"**

\[ \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(	ext{Mcm}(\text{WbAck}(), a), mcx, mcy), \text{cms}, \text{data}) \]

if \((mcx \neq mcy)\)
where \(\text{Cell}(a, v, \text{WbPending}()) = \text{data}[\text{hash}(a)]\)

\[ \Rightarrow \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(-, mcx, mcx), \text{cms}, \text{data}[\text{hash}(a):=\text{Cell}(a, v, \text{Clean}())]) \]

**Rule "WS28 Write Back Flush"**

\[ \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(	ext{Mcm}(\text{WbAckFlush}(), a), mcx, mcy), \text{cms}, \text{data}) \]

if \((mcx \neq mcy)\)
where \(\text{Cell}(a, v, \text{WbPending}()) = \text{data}[\text{hash}(a)]\)

\[ \Rightarrow \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(-, mcx, mcx), \text{cms}, \text{data}[\text{hash}(a):=\text{Invalid}()]) \]

**Rule "WS29 Clean Purge Request"**

\[ \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(	ext{Mcm}(\text{PurgeReq}(), a), mcx, mcy), \text{Cms}(-, cmx, cmy), \text{data}) \]

if \(((mcx \neq mcy) \&\& (cmx == cmy))\)
where \(\text{Cell}(a, -, \text{Clean}()) = \text{data}[\text{hash}(a)]\)

\[ \Rightarrow \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(-, mcx, mcx), \text{Cms}(	ext{Cmm}(\text{Purged}(), a), \text{not}(cmx), cmy), \text{data}[\text{hash}(a):=\text{Invalid}()]) \]

**Rule "WS30 Dirty Purge Request"**

\[ \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(	ext{Mcm}(\text{PurgeReq}(), a), mcx, mcy), \text{Cms}(-, cmx, cmy), \text{data}) \]

if \(((mcx \neq mcy) \&\& (cmx == cmy))\)
where \(\text{Cell}(a, v, \text{Dirty}()) = \text{data}[\text{hash}(a)]\)

\[ \Rightarrow \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(-, mcx, mcx), \text{Cms}(	ext{Cmm}(\text{Wb}(v), a), \text{not}(cmx), cmy), \text{data}[\text{hash}(a):=\text{Cell}(a, v, \text{WbPending}())]) \]

**Rule "WS31 + WS32 CMiss or WbPending Purge Request"**

\[ \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(	ext{Mcm}(\text{PurgeReq}(), a), mcx, mcy), \text{cms}, \text{data}) \]

if \(((mcx \neq mcy) \&\& (\text{MorWBP}(cs))\)
where \(cs = \text{data}[\text{hash}(a)]\)

\[ \Rightarrow \text{Sache}(\text{pcs}, \text{cps}, \text{Mcs}(-, mcx, mcx), \text{cms}, \text{data}) \]
4.5.2 Memory Controller of Cachet-WriterPush

In Cachet-WriterPush, there are three types of requests that can arrive at memory: a cache request, a writeback, and a purge acknowledgement. The following four rules come from the protocol directly. After these four, we get into some more interesting rules that require more states.

**Rule "WM1 Cache Request While Stable"**

\[
\text{SysMem(Ins(InM(CacheReq(a,ws),inx,iny),Outs(-,outx, outy),memdata,Nil))} \\
\text{if ((inx \neq iny) \&\& (outx == outy)) \&\& (notinlist(s,ws)) \} \\
\text{where MemCell(v,s) = memdata[a]} \\
\text{Stable(s0,s1,s2,s3) = s} \\
\text{\Rightarrow SysMem(Ins(-,inx,inx),Outs(OutM(Cache(v),a,ws),not(outx),outy),} \\
\text{memdata[a:=MemCell(v,addlist(s,ws))],Nil))}
\]

**Rule "WM2 Cache Request While Transient"**

\[
\text{SysMem(Ins(InM(CacheReq(a,ws),inx,iny),Outs(-,outx, outy),memdata,Nil))} \\
\text{if ((inx \neq iny) \&\& (outx == outy)) \&\& (notinlist(s,ws)) \} \\
\text{where MemCell(-,s) = memdata[a]} \\
\text{Transient(s0,s1,s2,s3,v,wsite) = s} \\
\text{\Rightarrow SysMem(Ins(-,inx,inx),Outs(OutM(Retry(a,ws),not(outx),outy),} \\
\text{memdata,Nil))}
\]

**Rule "WM4 Fast Write Back"**

\[
\text{SysMem(Ins(InM(Wb(v),a,ws),inx,iny),Outs(-,outx, outy),memdata,Nil())} \\
\text{if ((inx \neq iny) \&\& (outx == outy)) \&\& (issolitary(s,ws)) \} \\
\text{where MemCell(v,s) = memdata[a]} \\
\text{Stable(s0,s1,s2,s3) = s} \\
\text{\Rightarrow SysMem(Ins(-,inx,inx),Outs(OutM(WbAck(v,a,ws),not(outx),outy),} \\
\text{memdata[a:=MemCell(v,Stable(s0,s1,s2,s3))],Nil))}
\]

**Rule "WM5 Write Back while Transient"**

\[
\text{SysMem(Ins(InM(Wb(-),a,ws),inx,iny),Outs(-,outx, outy),memdata, Nil())} \\
\text{if ((inx \neq iny) \&\& (outx == outy)) \&\& (nonsolitary(s,ws)) \} \\
\text{where MemCell(-,s) = memdata[a]} \\
\text{Transient(s0,s1,s2,s3,v,ss) = s} \\
\text{\Rightarrow SysMem(Ins(-,inx,inx),Outs(OutM(WbAckFlush(v),a,ws),not(outx),outy),} \\
\text{memdata[a:=MemCell(-,rmlistT(s,ws))],Nil())}
\]

The case when memory receives a writeback request and is stable with more than one site in the list, is illustrated with the next three rules. Before the writeback can complete, purge messages must be sent to all sites holding a copy of the location. The model in this paper send a purge request to all three of the other sites, and if any of those sites does not have the location, then it ignores the purge request. Since only one message can be put on the output port of the memory module per rule, temporary states were created for the memory to enter. These states only exist so that we can send multiple messages (in this case, three
messages) in response to a single input message. Since one input writeback message spawns three purge requests, two temporary states are used to send the three messages. The first message is sent when the writeback arrives, and the state is changed to Purgings. This indicates that there is still two purging messages left to be sent. When the output port is ready for data again, the second purge message is sent and the Purgings state becomes the Purging state. The Purging state indicates that there is a single purge message remaining to be sent. When the output port is again free, the Purging state goes back to the default state of Nil, and sends out the third purge request message.

**Rule "WM3A Write Back"**
SysMem(Ins(InM(Wb(v),a,ws),inx,iny),Outs(,outx,outy),memdata, Nil())
   if ( ((inx != iny) && (outx == outy)) && (nonsolitary(s,ws)) )
   where MemCell(,s) = memdata[a]
   Stable(s0,s1,s2,s3) = s
===>SysMem(Ins(,inx,iny),Outs(OutM(PurgeReq0,a,(ws==0)?1:0),not(outx),outy),memdata[a:=MemCell(,Transient(s0,s1,s2,s3,v,ws))],
Purgings(a,ws))

**Rule "WM3B Purgings temp state"**
SysMem(ins,Outs(,outx, outy),memdata,Purgings(a,ws))
   if (outx == outy)
===>SysMem(ins,Outs(OutM(PurgeReq0,a,(ws==0)?2:(ws==1)?2:1)),
   not(outx,outy),memdata,Purging(a,ws))

**Rule "WM3C Purging temp state"**
SysMem(Ins(-,inx,-),Outs(-,outx, outy),memdata,Purging(a,ws))
   if (outx == outy)
===>SysMem(Ins(-,inx,inx),Outs(OutM(PurgeReq0,a,(ws==3)?2:3),
   not(outx),outy),memdata,Nil())

The case when a writeback request gets to memory while in the transient state, and it is the last site in the list, also causes multiple messages to be sent in response to a single input message. Both a writeback flush and a writeback acknowledgement must be sent. As before, when we wanted to send three messages in response to a single input message, we had to use temporary states. This time, with two messages to send, we need only one temporary state.

**Rule "WM6A Flush & WbAck"**
SysMem(Ins(InM(Wb(-),a,ws),inx,iny),Outs(-,outx, outy),memdata, Nil())
   if ( ((inx != iny) && (outx == outy)) && (issolitary(s,ws)) )
   where MemCell(,s) = memdata[a]
   Transient(s0,s1,s2,s3,v,ss) = s
===>SysMem(Ins(-,inx,iny),Outs(OutM(WbAckFlush0,a,ws),not(outx),outy),memdata[a:=MemCell(,makesolitary(ns))],WbAcking(a,ns))

44
Rule "WM6B WbRouting ack state"
SysMem(Ins(-,inx,-),Outs(-,outx, outy), memdata, WbRouting(a, ns))
  if (outx == outy)
  ==> SysMem(Ins(-,inx,inx), Outs(OutM(WbAck(), a, ns), not(outx), outy),
            memdata, Nil())

The remaining rules given for the cache controller again follow from the Protocol in Figure 3.7, on a 1-to-1 basis.

Rule "WM7 Purged while Stable"
SysMem(Ins(InM(Purged(),a,ws),inx,iny), outs, memdata, Nil())
  if ( (inx != iny) && (inlist(s,ws)) )
    where MemCell(-,s) = memdata[a]
    Stable(s0,s1,s2,s3) = s
  ==> SysMem(Ins(-,inx,inx), outs, memdata[a:=MemCell(v,rmlist(s,ws))], Nil())

Rule "WM8 Purged while Transient"
SysMem(Ins(InM(Purged(),a,ws),inx,iny), outs, memdata, Nil())
  if ( (inx != iny) && (non solitary(s,ws)) )
    where MemCell(-,s) = memdata[a]
    Transient(s0,s1,s2,s3,v,ss) = s
  ==> SysMem(Ins(-,inx,inx), outs, memdata[a:=MemCell(v,rmlistT(s,ws))], Nil())

Rule "WM9 Writeback Completes"
SysMem(Ins(InM(Purged(),a,ws),inx,iny), Outs(-,outx, outy), memdata, Nil())
  if ( ((inx != iny) && (outx == outy)) && (issolitary(s,ws)) )
    where MemCell(v,s) = memdata[a]
    Transient(s0,s1,s2,s3,v,ns) = s
  ==> SysMem(Ins(-,inx,inx), Outs(OutM(WbAck(),a,ns),not(outx),outy),
            memdata[a:=MemCell(v,makesolitary(ws))], Nil())

4.6 Cachet-Migratory

This next section describes the rules implementing the cache and memory interfaces for the Cachet-Migratory protocol. First, the rules of the Sache Controller will be explained, and following that will come the rules for the Memory Controller.
4.6.1 Sache Controller Rules of Cachet-Migratory

Just as Cachet-WriterPush, the rule for Clean Reconcile is such that we keep the location in sache in a clean state. However, a change from both of the other protocols is the action when we get a dirty commit in Migratory: we do nothing. We retire the commit without sending any messages to memory or changing the state of the sache cell.

Rule "MS13 + MS14 + MS16 Clean or Dirty or CMiss Commit"
Sache(Pcs(Pem(t, Commit()),a),pcx,pcy),Cps(-,cpx,cpy),mcs,cms,data)
if ( ((pcx! =pcy) & (cpx == cp’y)) & (CorDorM(cs))
where cs = data[has(a)]
===>Sache(Pcs(-,pcx,pcx),Cps(Cpm(t),not(cpx),cpy),mcs,cms,data)

There are two types of messages the processor can get from memory in Cachet-Migratory, cache messages and purge-request messages, given below. Since these differ from the rules of Cachet-Base which process messages from memory, they are given below.

Rule "MS21 CacheReq Response"
Sache(pcs,cps,Mcs(Mcm(Cache(v),a),mcx,mcy),cms, data)
if (mcx != mcy)
where Cell(a,-,CachePending() = data[has(a)]
===>Sache(pcs,cps,Mcs(-,mcx,mcy),cms, data[has(a):=Cell(a,v,Clean())])

Rule "MS22 Retry"
Sache(pcs,cps,Mcs(Mcm(Retry(),a),mcx,mcy),cms, data)
if (mcx != mcy)
where Cell(a,-,CachePending() = data[has(a)]
===>Sache(pcs,cps,Mcs(-,mcx,mcx),cms, data[has(a):=Invalid()])

Rule "MS23 Clean Purge Request"
Sache(pcs,cps,Mcs(Mcm(PurgeReq(),a),mcx,mcy),Cms(-,cmx,cm),data)
if ((mcx != mcy) & (cmx == cmx))
where Cell(a,-,Clean()) = data[has(a)]
===>Sache(pcs,cps,Mcs(-,mcx,mcx),Cms(Cmm(Purged(),a),
not(cm),cm),data[has(a):=Invalid()])

Rule "MS24 Dirty Purge Request"
Sache(pcs,cps,Mcs(Mcm(PurgeReq(),a),mcx,mcy),Cms(-,cmx,cm),data)
if ((mcx != mcy) & (cmx == cmx))
where Cell(a,v,Dirty()) = data[has(a)]
===>Sache(pcs,cps,Mcs(-,mcx,mcx),Cms(Cmm(Wb(v),a),not(cm),cm),
data[has(a):=Invalid()])
4.6.2 Memory Controller of Cachet-Migratory

The three types of messages that memory can receive in Cachet-Migratory are cache request messages, writeback messages, and purge acknowledgement messages. They follow directly from the protocol given in Figure 3.10.

**Rule "MM1 Cache Request"**

SysMem(Ins(InM(CacheReq(),a,ws),inx,iny), Outs(-,outx,outy), memdata,Nil)
if ((inx != iny) \&\& (outx == outy))
where MemCell(v,Noone()) = memdata[a]

implies SysMem(Ins(-,inx,inx), Outs(OutM(Cache(v),a,ws),not(outx),outy),
memdata[a:=MemCell(v,Exclusive(ws))],Nil)

**Rule "MM2 Bump Out"**

SysMem(Ins(InM(CacheReq(),a,ws),inx,iny), Outs(-,outx,outy), memdata,Nil)
if ((inx != iny) \&\& (outx == outy))
where MemCell(v,Exclusive(holder)) = memdata[a]

implies SysMem(Ins(-,inx,inx), Outs(OutM(PurgeReq(),a,holder),not(outx),outy),
memdata[a:=MemCell(v,MigTrans(holder, ws))],Nil)

**Rule "MM3 Cache Request while MigTrans"**

SysMem(Ins(InM(CacheReq(),a,ws),inx,iny), Outs(-,outx,outy), memdata,Nil)
if ((inx != iny) \&\& (outx == outy))
where MemCell(v,Transient(-,-,-,-,-)) = memdata[a]

implies SysMem(Ins(-,inx,inx), Outs(OutM(Retry(),a,ws),not(outx),outy),
memdata,Nil)

**Rule "MM4 Purged while Exclusive"**

SysMem(Ins(InM(Purged(),a,ws),inx,iny),outs,memdata,Nil)
if (inx != iny)
where MemCell(v,Exclusive(ws)) = memdata[a]

implies SysMem(Ins(-,inx,inx),outs,memdata[a:=MemCell(v,Noone())],Nil)

**Rule "MM5 Purged while MigTrans"**

SysMem(Ins(InM(Purged(),a,ws),inx,iny), Outs(-,outx,outy), memdata,Nil)
if ((inx != iny) \&\& (outx == outy))
where MemCell(v,MigTrans(ws,ns)) = memdata[a]

implies SysMem(Ins(-,inx,inx), Outs(OutM(Cache(v),a,ns),not(outx),outy),
memdata[a:=MemCell(v,Exclusive(ns))],Nil)
Rule "MM6 Write Back while Exclusive"
SysMem(Ins(InM(Wb(v),a,ws),inx,iny),outs,memdata, Nil())
   if (inx != iny)
      where MemCell(Exclusive(ws)) = memdata[a]
==>SysMem(Ins(-,inx,inx),outs,memdata[a:=MemCell(v,Noone())],Nil())

Rule "MM7 Write Back while MigTrans"
SysMem(Ins(InM(Wb(v),a,ws),inx,iny),Outs(-,outx,outy),memdata, Nil())
   if ((inx != iny) && (outx == outy))
      where MemCell(MigTrans(ws,ns)) = memdata[a]
==>SysMem(Ins(-,inx,inx),Outs(OutM(Cache(v),a,ns),not(outx),outy),
         memdata[a:=MemCell(v,Exclusive(ns))],Nil())
Chapter 5

Conclusions

In this Chapter comes the conclusions drawn from the compilation of the translated Computer Readable TRS rules, and what further work should follow.

In this research we set out to show that Cachet is a synthesizable cache coherence protocol. We have proven that we can take the high level description of Cachet, expressed in a TRS by Xiaowei Shen, and rewrite it to make it suitable for compilation on the TRAC compiler. After the translation of Xiaowei's Cachet rules, the rules had to be compiled in James Hoe's TRAC compiler. Overcoming a few large bugs and a number of minor ones, the Computer Readable TRS was successfully compiled on TRAC, and output RTL Verilog code. This RTL output can next be simulated and synthesized using commercial tools, as will be discussed below. The work of this thesis has expanded the understanding of the Cachet protocol, by exploring how well it can be implemented in hardware.

In Figure 5.1, we illustrate the size, in kilobytes, of the RTL Verilog files created by running the appropriate set of rules through the TRAC compiler. In the cases when it is a sache or group of rules, only the sache definitions were used in the compilation, and similarly for a memory group of rules only the memory definitions were used in the compilation.

<table>
<thead>
<tr>
<th></th>
<th>Sache (kB)</th>
<th>Memory (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>650.2</td>
<td>1136.2</td>
</tr>
<tr>
<td>WriterPush</td>
<td>657.8</td>
<td>1365.2</td>
</tr>
<tr>
<td>Migratory</td>
<td>634.8</td>
<td>1193.4</td>
</tr>
</tbody>
</table>

Figure 5.1 Size of RTL Code Generated

There is also the issue of the microprotocols being merged. While a partial merging was described in “CACHET: An Adaptive Cache Coherence Protocol For Distributed Shared Memory Systems” [10], a complete merging of the protocols which would be necessary for an implementable overall Cachet protocol has not been done. This would require taking the description of a three-way merging, as described in “Design and Verification of Adaptive Cache Coherence Protocols” [8],
and then removing all the voluntary protocol rules. However, this removal of voluntary rules makes for difficult reasoning that must be done about the protocol, to ensure that it is still correct. This reasoning is very rigorous and must be done carefully, as it is extremely easy to make mistakes in protocols. Issues of when to change a location from one protocol to another were handled abstractly in the description of a three-way merging of the microprotocols; however, without voluntary rules, it must be made explicit where and when these changes will occur. Will they be based off of some state that is kept on the access pattern to a location? Is that state worth keeping? If a data location is 4 bytes and you need to keep 2 bytes of state information related to recent accesses of that location, is such a 50% overhead worthwhile? What if 8 bytes of state were required to implement the state that Cachet needs... would that be worth it?

This work has shown that the rules can be translated and compiled, and the Verilog RTL output of that compiler is ready for the next steps: simulation, synthesis and manufacturing.

There are two additional steps that logically follow the work of this thesis, which have not been undertaken. The first of these two steps would be to take the Verilog RTL output of the TRAC compiler, and send it to a Real Circuit Simulator, to simulate the running of these cache coherence protocols. This Simulator will tell us how well the RTL code would perform, if we built actual working hardware and ran the Cachet protocol on that hardware. The second of these two steps would be to synthesize and eventually manufacture working hardware that runs these cache coherence protocols. Simulatability implies synthesizability; synthesizability implies constructability; constructability implies correctness. Thus, if we can simulate this Verilog RTL, it would be a proof of correctness of the Cachet cache coherence protocols.

Something to be considered by future researchers is scheduling of rule priorities: when two rules are free to fire, which will go? The answer is that scheduling should be prioritized to make imperative rules get a higher priority than directive rules. This means that in the cache controller, all rules that process messages from memory should be checked with a higher scheduling priority than those relating to processor issued instruction messages.

If Cachet proves to be successful in simulation and synthesis, then it would be implemented in a hardware system, and that system would run programs faster: the unattainable goal of ‘faster computers’ would be closer to being realized.
Appendices

Appendix A: Implementable Protocol of Cachet-Base

**Processor Rules of Base**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cstate</th>
<th>MSG-&gt;Mem, Action</th>
<th>Next Cstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS1 &lt;LoadL,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>BS2 &lt;LoadL,a&gt;</td>
<td>Cell(a,v,Dirty)</td>
<td>retire</td>
<td>Cell(a,v,Dirty)</td>
</tr>
<tr>
<td>BS3 &lt;LoadL,a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>BS4 &lt;LoadL,a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>BS5 &lt;LoadL,a&gt;</td>
<td>a not in cache, cache not full</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>BS6 &lt;LoadL,a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,-,Clean)</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>BS7 &lt;LoadL,a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,v,Dirty)</td>
<td>&lt;Wb(v),a1&gt; &amp; stall</td>
<td>Cell(a1,v,WbPending)</td>
</tr>
<tr>
<td>BS8 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Dirty)</td>
</tr>
<tr>
<td>BS9 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,v,Dirty)</td>
<td>retire</td>
<td>Cell(a,v,Dirty)</td>
</tr>
<tr>
<td>BS10 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>BS11 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>BS12 &lt;StoreL(v),a&gt;</td>
<td>a not in cache, cache not full</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>BS13 &lt;StoreL(v),a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,v,Clean)</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>BS14 &lt;StoreL(v),a&gt;</td>
<td>a not in cache, cache is full, has Cell(a1,v,Dirty)</td>
<td>&lt;Wb(v),a1&gt; &amp; stall</td>
<td>Cell(a1,v,WbPending)</td>
</tr>
<tr>
<td>BS15 &lt;Commit,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>BS16 &lt;Commit,a&gt;</td>
<td>Cell(a,v,Dirty)</td>
<td>&lt;Wb(v),a&gt;, stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>BS17 &lt;Commit,a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>BS18 &lt;Commit,a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>BS19 &lt;Commit,a&gt;</td>
<td>a not in cache</td>
<td>retire</td>
<td>a not in cache</td>
</tr>
<tr>
<td>BS20 &lt;Reconcile,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell&lt;--Invalid</td>
</tr>
<tr>
<td>BS21 &lt;Reconcile,a&gt;</td>
<td>Cell(a,v,Dirty)</td>
<td>retire</td>
<td>Cell(a,v,Dirty)</td>
</tr>
<tr>
<td>BS22 &lt;Reconcile,a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>BS23 &lt;Reconcile,a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>BS24 &lt;Reconcile,a&gt;</td>
<td>a not in cache</td>
<td>retire</td>
<td>a not in cache</td>
</tr>
</tbody>
</table>
### Cache-Engine Rules of Base

<table>
<thead>
<tr>
<th>Msg from Mem</th>
<th>Cstate</th>
<th>Action</th>
<th>Next Cstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS25</td>
<td><code>&lt;Cache(v),a&gt;</code></td>
<td>Cell(a,-,CachePending)</td>
<td>uninstall</td>
</tr>
<tr>
<td>BS26</td>
<td><code>&lt;WbAck,a&gt;</code></td>
<td>Cell(a,v,WbPending)</td>
<td>uninstall</td>
</tr>
</tbody>
</table>

### Memory-Engine Rules of Base

<table>
<thead>
<tr>
<th>Msg from id</th>
<th>Mstate</th>
<th>Msg to id</th>
<th>Next Mstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM1</td>
<td><code>&lt;CacheReq,a&gt;</code></td>
<td>Cell(a,v,_)</td>
<td><code>&lt;Cache(v),a&gt;</code></td>
</tr>
<tr>
<td>BM2</td>
<td><code>&lt;Wb(v),a&gt;</code></td>
<td>Cell(a,<em>,</em>)</td>
<td><code>&lt;WbAck,a&gt;</code></td>
</tr>
</tbody>
</table>
Appendix B: Implementable Protocol of Cachet-WriterPush

Processor Rules of WP

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cstate</th>
<th>MSGs---&gt;Mem, Action</th>
<th>Next Cstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS1 &lt;LoadL,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>WS2 &lt;LoadL,a&gt;</td>
<td>Cell(a,v,Dirty)</td>
<td>retire</td>
<td>Cell(a,v,Dirty)</td>
</tr>
<tr>
<td>WS3 &lt;LoadL,a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>WS4 &lt;LoadL,a&gt;</td>
<td>Cell(a,-, CachePending)</td>
<td>stall</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td>WS5 &lt;LoadL,a&gt;</td>
<td>a not in cache,</td>
<td>&lt;CacheReq,a&gt; &amp;</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td></td>
<td>cache not full</td>
<td>stall</td>
<td></td>
</tr>
<tr>
<td>WS6 &lt;LoadL,a&gt;</td>
<td>a not in cache,</td>
<td>&lt;Purged,a1&gt; &amp;</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td></td>
<td>cache is full,</td>
<td>stall</td>
<td></td>
</tr>
<tr>
<td></td>
<td>has Cell(a1,-, Clean)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WS7 &lt;LoadL,a&gt;</td>
<td>a not in cache,</td>
<td>&lt;Wb(v),a1&gt; &amp;</td>
<td>Cell(a1,v,WbPending)</td>
</tr>
<tr>
<td></td>
<td>cache is full,</td>
<td>stall</td>
<td></td>
</tr>
<tr>
<td></td>
<td>has Cell(a1,v,Dirty)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WS8 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Dirty)</td>
</tr>
<tr>
<td>WS9 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,v,Dirty)</td>
<td>retire</td>
<td>Cell(a,v,Dirty)</td>
</tr>
<tr>
<td>WS10 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>WS11 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,-, CachePending)</td>
<td>stall</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td>WS12 &lt;StoreL(v),a&gt;</td>
<td>a not in cache,</td>
<td>&lt;CacheReq,a&gt; &amp;</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td></td>
<td>cache not full</td>
<td>stall</td>
<td></td>
</tr>
<tr>
<td>WS13 &lt;StoreL(v),a&gt;</td>
<td>a not in cache,</td>
<td>&lt;Purged,a1&gt; &amp;</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td></td>
<td>cache is full,</td>
<td>stall</td>
<td></td>
</tr>
<tr>
<td></td>
<td>has Cell(a1,v,Clean)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WS14 &lt;StoreL(v),a&gt;</td>
<td>a not in cache,</td>
<td>&lt;Wb(v),a1&gt; &amp;</td>
<td>Cell(a1,v,WbPending)</td>
</tr>
<tr>
<td></td>
<td>cache is full,</td>
<td>stall</td>
<td></td>
</tr>
<tr>
<td></td>
<td>has Cell(a1,v,Dirty)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WS15 &lt;Commit,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>WS16 &lt;Commit,a&gt;</td>
<td>Cell(a,v,Dirty)</td>
<td>&lt;Wb(v),a&gt; &amp;</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>WS17 &lt;Commit,a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>WS18 &lt;Commit,a&gt;</td>
<td>Cell(a,-, CachePending)</td>
<td>stall</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td>WS19 &lt;Commit,a&gt;</td>
<td>a not in cache</td>
<td>retire</td>
<td>a not in cache</td>
</tr>
<tr>
<td>WS20 &lt;Reconcile,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>WS21 &lt;Reconcile,a&gt;</td>
<td>Cell(a,v,Dirty)</td>
<td>retire</td>
<td>Cell(a,v,Dirty)</td>
</tr>
<tr>
<td>WS22 &lt;Reconcile,a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>stall</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>WS23 &lt;Reconcile,a&gt;</td>
<td>Cell(a,-, CachePending)</td>
<td>stall</td>
<td>Cell(a,-, CachePending)</td>
</tr>
<tr>
<td>WS24 &lt;Reconcile,a&gt;</td>
<td>a not in cache</td>
<td>retire</td>
<td>a not in cache</td>
</tr>
</tbody>
</table>
### Cache-Engine Rules of WP

<table>
<thead>
<tr>
<th>Msg from Mem</th>
<th>Cstate</th>
<th>Action</th>
<th>Next Cstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS25 &lt;Cache(v),a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>uninstall</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>WS26 &lt;Retry,a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>uninstall</td>
<td>Cell&lt;-Invalid</td>
</tr>
<tr>
<td>WS27 &lt;WbAck,a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>uninstall</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>WS28 &lt;FlushAck,a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>uninstall</td>
<td>Cell&lt;-Invalid</td>
</tr>
<tr>
<td>WS29 &lt;PurgeReq,a&gt;</td>
<td>Cell(a,-,Clean)</td>
<td>&lt;Purged,a&gt; Cell&lt;-Invalid</td>
<td></td>
</tr>
<tr>
<td>WS30 &lt;PurgeReq,a&gt;</td>
<td>Cell(a,v,Dirty)</td>
<td>&lt;Wb(v),a&gt;  Cell(a,v,WbPending)</td>
<td></td>
</tr>
<tr>
<td>WS31 &lt;PurgeReq,a&gt;</td>
<td>Cell(a,v,WbPending)</td>
<td>none</td>
<td>Cell(a,v,WbPending)</td>
</tr>
<tr>
<td>WS32 &lt;PurgeReq,a&gt;</td>
<td>a not in cache</td>
<td>none</td>
<td>a not in cache</td>
</tr>
</tbody>
</table>

### Memory-Engine Rules of WP

<table>
<thead>
<tr>
<th>Msg from id</th>
<th>Mstate</th>
<th>Msg(s) to send</th>
<th>Next Mstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>WM1 &lt;CacheReq,a&gt;</td>
<td>Cell(a,v,C[dir]) &amp; id not in dir</td>
<td>&lt;Cache(v),a&gt; -&gt; id</td>
<td>Cell(a,v,C[id</td>
</tr>
<tr>
<td>WM2 &lt;CacheReq,a&gt;</td>
<td>Cell(a,-,T[dir,sm]) &amp; id not in dir</td>
<td>&lt;Retry,a&gt; -&gt; id</td>
<td>Cell(a,-, T[dir,sm])</td>
</tr>
<tr>
<td>WM3 &lt;Wb(v),a&gt;</td>
<td>Cell(a,-,C[id</td>
<td>dir]) &amp; dir nonempty</td>
<td>&lt;PurgeReq,a&gt; -&gt; dir</td>
</tr>
<tr>
<td>WM4 &lt;Wb(v),a&gt;</td>
<td>Cell(a,-,C[id])</td>
<td>&lt;WbAck,a&gt; -&gt; id</td>
<td>Cell(a,v,C[id])</td>
</tr>
<tr>
<td>WM5 &lt;Wb(v),a&gt;</td>
<td>Cell(a,-,T[id</td>
<td>dir,sm]) &amp; dir nonempty</td>
<td>&lt;FlushAck,a&gt; -&gt; id</td>
</tr>
<tr>
<td>WM6 &lt;Wb(v),a&gt;</td>
<td>Cell(a,-,T[id,(id1,v)])</td>
<td>&lt;FlushAck,a&gt; -&gt; id</td>
<td>Cell(a,v,C[id1])</td>
</tr>
<tr>
<td>WM7 &lt;Purged,a&gt;</td>
<td>Cell(a,-,C[id</td>
<td>dir])</td>
<td>none</td>
</tr>
<tr>
<td>WM8 &lt;Purged,a&gt;</td>
<td>Cell(a,-,T[id</td>
<td>dir,sm]) &amp; dir nonempty</td>
<td>none</td>
</tr>
<tr>
<td>WM9 &lt;Purged,a&gt;</td>
<td>Cell(a,-,T[id,(id1,v)])</td>
<td>&lt;WbAck,a&gt; -&gt; id</td>
<td>Cell(a,v,C[id1])</td>
</tr>
</tbody>
</table>

54
Appendix C: Implementable Protocol of Cachet-Migratory

Processor Rules of Migratory

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cstate</th>
<th>MSGs-&gt;Mem, Action</th>
<th>Next Cstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS1 &lt;LoadL,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>MS2 &lt;LoadL,a&gt;</td>
<td>Cell(a,v,Dirt)y</td>
<td>retire</td>
<td>Cell(a,v,Dirt)y</td>
</tr>
<tr>
<td>MS3 &lt;LoadL,a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>MS4 &lt;LoadL,a&gt;</td>
<td>a not in cache,</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td></td>
<td>cache not full</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS5 &lt;LoadL,a&gt;</td>
<td>a not in cache,</td>
<td>&lt;Purged,a1&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td></td>
<td>cache is full,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>has Cell(a1,-,Clean)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS6 &lt;LoadL,a&gt;</td>
<td>a not in cache,</td>
<td>&lt;Wb(v),a1&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td></td>
<td>cache is full,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>has Cell(a1,v,Dirty)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS7 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Dirt)y</td>
</tr>
<tr>
<td>MS8 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,v,Dirt)y</td>
<td>retire</td>
<td>Cell(a,v,Dirt)y</td>
</tr>
<tr>
<td>MS9 &lt;StoreL(v),a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>MS10 &lt;StoreL(v),a&gt;</td>
<td>a not in cache,</td>
<td>&lt;CacheReq,a&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td></td>
<td>cache not full</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS11 &lt;StoreL(v),a&gt;</td>
<td>a not in cache,</td>
<td>&lt;Purged,a1&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td></td>
<td>cache is full,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>has Cell(a1,v,Clean)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS12 &lt;StoreL(v),a&gt;</td>
<td>a not in cache,</td>
<td>&lt;Wb(v),a1&gt; &amp; stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td></td>
<td>cache is full,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>has Cell(a1,v,Dirty)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MS13 &lt;Commit,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>MS14 &lt;Commit,a&gt;</td>
<td>Cell(a,v,Dirt)y</td>
<td>retire</td>
<td>Cell(a,v,Dirt)y</td>
</tr>
<tr>
<td>MS15 &lt;Commit,a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>MS16 &lt;Commit,a&gt;</td>
<td>a not in cache</td>
<td>retire</td>
<td>a not in cache</td>
</tr>
<tr>
<td>MS17 &lt;Reconcile,a&gt;</td>
<td>Cell(a,v,Clean)</td>
<td>retire</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>MS18 &lt;Reconcile,a&gt;</td>
<td>Cell(a,v,Dirt)y</td>
<td>retire</td>
<td>Cell(a,v,Dirt)y</td>
</tr>
<tr>
<td>MS19 &lt;Reconcile,a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>stall</td>
<td>Cell(a,-,CachePending)</td>
</tr>
<tr>
<td>MS20 &lt;Reconcile,a&gt;</td>
<td>a not in cache</td>
<td>retire</td>
<td>a not in cache</td>
</tr>
</tbody>
</table>

Cache-Engine Rules of Migratory

<table>
<thead>
<tr>
<th>Msg from Mem</th>
<th>Cstate</th>
<th>Action</th>
<th>Next Cstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS21 &lt;Cache(v),a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>uninstall</td>
<td>Cell(a,v,Clean)</td>
</tr>
<tr>
<td>MS22 &lt;Retry,a&gt;</td>
<td>Cell(a,-,CachePending)</td>
<td>uninstall</td>
<td>Cell&lt;-Invalid</td>
</tr>
<tr>
<td>MS23 &lt;PurgeReq,a&gt;</td>
<td>Cell(a,-,Clean)</td>
<td>&lt;Purged,a&gt;</td>
<td>Cell&lt;-Invalid</td>
</tr>
<tr>
<td>MS24 &lt;PurgeReq,a&gt;</td>
<td>Cell(a,v,Dirt)y</td>
<td>&lt;Wb(v),a&gt;</td>
<td>Cell&lt;-Invalid</td>
</tr>
<tr>
<td>MS25 &lt;PurgeReq,a&gt;</td>
<td>a not in cache</td>
<td>none</td>
<td>a not in cache</td>
</tr>
</tbody>
</table>
Memory-Engine Rules of Migratory

<table>
<thead>
<tr>
<th>Msg from id</th>
<th>Mstate</th>
<th>Msg(s) to send</th>
<th>Next Mstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM1 &lt;CacheReq&gt;a&gt;</td>
<td>Cell(a,v,NOONE)</td>
<td>&lt;Cache(v),a&gt; -&gt; id</td>
<td>Cell(a,v,C[id])</td>
</tr>
<tr>
<td>MM2 &lt;CacheReq,a&gt;</td>
<td>Cell(a,v,C[id1]),id!=id1</td>
<td>&lt;PurgeReq,a&gt; -&gt; id</td>
<td>Cell(a,v,T[id1,id2])</td>
</tr>
<tr>
<td>MM3 &lt;CacheReq,a&gt;</td>
<td>Cell(a,v,T[id1,id2])</td>
<td>&lt;Retry,a&gt; -&gt; id</td>
<td>Cell(a,v,T[id1,id2])</td>
</tr>
<tr>
<td>MM4 &lt;Purged,a&gt;</td>
<td>Cell(a,v,C[id])</td>
<td>none</td>
<td>Cell(a,v,NOONE)</td>
</tr>
<tr>
<td>MM5 &lt;Purged,a&gt;</td>
<td>Cell(a,v,T[id1])</td>
<td>&lt;Cache(v),a&gt; -&gt; id1</td>
<td>Cell(a,v,C[id1])</td>
</tr>
<tr>
<td>MM6 &lt;Wb(v),a&gt;</td>
<td>Cell(a,v,C[id])</td>
<td>none</td>
<td>Cell(a,v,NOONE)</td>
</tr>
<tr>
<td>MM7 &lt;Wb(v),a&gt;</td>
<td>Cell(a,v,T[id1])</td>
<td>&lt;Cache(v),a&gt; -&gt; id1</td>
<td>Cell(a,v,C[id1])</td>
</tr>
</tbody>
</table>
Appendix D: State Descriptions for Cachet

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOP</td>
<td>Top(SITE,SITE,SITE,SITE,MSITE)</td>
</tr>
<tr>
<td>SITE</td>
<td>Site(Proc(),SACHE,PMBuf(),MPBuf())</td>
</tr>
<tr>
<td>MSITE</td>
<td>MSite(MemInb(),MemOutb(),SYSMEM)</td>
</tr>
<tr>
<td>SACHE</td>
<td>Sache(PCS,CPS,MCS,CMS,DATA)</td>
</tr>
<tr>
<td>DATA</td>
<td>Array [ADDR]CELL</td>
</tr>
<tr>
<td>CELL</td>
<td>Cell(ADDR, VAL, CELLTAG)</td>
</tr>
<tr>
<td>CELLLTAG</td>
<td>Clean()</td>
</tr>
<tr>
<td>PCS</td>
<td>Pcs(PCM,YN,YN)</td>
</tr>
<tr>
<td>PCM</td>
<td>Pcm(PROCTAG, INST, ADDR)</td>
</tr>
<tr>
<td>INST</td>
<td>LoadL()</td>
</tr>
<tr>
<td>PROCTAG</td>
<td>Bit[16]</td>
</tr>
<tr>
<td>CPS</td>
<td>Cps(CPM,YN,YN)</td>
</tr>
<tr>
<td>CPM</td>
<td>Cpm(PROCTAG, VALACK)</td>
</tr>
<tr>
<td>VALACK</td>
<td>Value(VAL)</td>
</tr>
<tr>
<td>MCS</td>
<td>Mcs(MCM,YN,YN)</td>
</tr>
<tr>
<td>MCM</td>
<td>Mcm(MCTAG, ADDR)</td>
</tr>
<tr>
<td>MCTAG</td>
<td>WbAck()</td>
</tr>
<tr>
<td>CMS</td>
<td>Cms(CMM,YN,YN)</td>
</tr>
<tr>
<td>CMM</td>
<td>Cmm(CMTAG, ADDR)</td>
</tr>
<tr>
<td>CMTAG</td>
<td>CacheReq()</td>
</tr>
<tr>
<td>SYSMEM</td>
<td>SysMem(INS, OUTS, MEMDATA, STATETAG)</td>
</tr>
<tr>
<td>STATETAG</td>
<td>Nil()</td>
</tr>
<tr>
<td>MEMDATA</td>
<td>Array [ADDR]MEMCELL</td>
</tr>
<tr>
<td>MEMCELL</td>
<td>MemCell(VAL, MCT)</td>
</tr>
<tr>
<td>MCT</td>
<td>Noone()</td>
</tr>
<tr>
<td>NUMSITE</td>
<td>Bit[2]</td>
</tr>
<tr>
<td>YN</td>
<td>Bit[1]</td>
</tr>
<tr>
<td>ADDR</td>
<td>Bit[16]</td>
</tr>
<tr>
<td>VAL</td>
<td>Bit[16]</td>
</tr>
<tr>
<td>INS</td>
<td>Ins(INM, YN, YN)</td>
</tr>
<tr>
<td>INM</td>
<td>InM(CMTAG, ADDR, NUMSITE)</td>
</tr>
<tr>
<td>OUTS</td>
<td>Outs(OUTM, YN, YN)</td>
</tr>
<tr>
<td>OUTM</td>
<td>OutM(MCTAG, ADDR, NUMSITE)</td>
</tr>
</tbody>
</table>
Appendix E: Computer Readable TRS of Cachet-Base

Rule "Base Clean or Dirty LoadL"
Sache(Pcs(Pcm(t,LoadL(),a),pcx,pcy),Cps(-,cpx,cpy),mcs,cms,data)  
  if ( ((pcx != pcy) && (cpx == cpy)) &&  
      (s == Clean()) || (s == Dirty()))  
  where Cell(a,v,s) = data[hash(a)]  
  ==>Sache(Ps(-,pcx,pcx),Cps(Cpm(t,Value(v)),cpz,cpy),mcs,cms,data)  
  where cpz = switch(cpx)  
    case 0: 1  
    case 1: 0

Rule "Base Invalid LoadL"
Sache(Pcs(Pcm(t,LoadL(),a),pcx,pcy),Cps,-,mcs,Cms(-,cmx,cmy),data)  
  if ( (pcx != pcy) && (cmx == cmy) )  
  where Invalid() = data[hash(a)]  
  ==>Sache(Ps(-,pcx,pcy),Cps,Cms(Cmm(CacheReq(a),cmz,cmy),  
      data[hash(a):=Cell(a,-,CachePending())])  
  where cmz = switch(cmx)  
    case 0: 1  
    case 1: 0

Rule "Base Clean Collision LoadL"
Sache(Pcs(Pcm(t,LoadL(),a),pcx,pcy),Cps,-,mcs,Cms(-,cmx,cmy),data)  
  if ( ((pcx != pcy) && (cmx == cmy)) && (a1 != a) )  
  where Cell(a1,v,Clean()) = data[hash(a)]  
  ==>Sache(Ps(-,pcx,pcy),Cps,Cms(Cmm(CacheReq(a),cmz,cmy),  
      data[hash(a):=Cell(a1,-,CachePending())])  
  where cmz = switch(cmx)  
    case 0: 1  
    case 1: 0

Rule "Base Dirty Collision LoadL"
Sache(Pcs(Pcm(t,LoadL(),a),pcx,pcy),Cps,-,mcs,Cms(-,cmx,cmy),data)  
  if ( ((pcx != pcy) && (cmx == cmy)) && (a1 != a) )  
  where Cell(a1,v,Dirty()) = data[hash(a)]  
  ==>Sache(Ps(-,pcx,pcy),Cps,Cms(Cmm(Wb(v),a1),cmz,cmy),  
      data[hash(a):=Cell(a1,v,WbPending())])  
  where cmz = switch(cmx)  
    case 0: 1  
    case 1: 0

Rule "Base Clean or Dirty StoreL"
Sache(Pcs(Pcm(t,StoreL(v),a),pcx,pcy),Cps,-,mcs,cms,data)  
  if ( ((pcx != pcy) && (cpx == cpy)) &&  
      (s == Clean()) || (s == Dirty()))  
  where Cell(a,-,s) = data[hash(a)]  
  ==>Sache(Ps(-,pcx,pcx),Cps(Cpm(t,Acko),cpz,cpy),mcs,cms,  
      data[hash(a):=Cell(a,v,Dirty())])  
  where cpz = switch(cpx)  
    case 0: 1  
    case 1: 0

59
Rule "Base Invalid StoreL"
Sache(Pcs(Pcm(t,StoreL(v),a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
  if ((pcx != pcy) && (cmx == cmy))
    where Invalid() = data[hash(a)]
==>Sache(Pcs(-,pcx,pcy),cps,mcs,Cms(Cmm(CacheReq(a),cmz,cmy),
    data[hash(a)]:=Cell(a,-,CachePending())))
    where cmz = switch(cmx)
      case 0: 1
      case 1: 0

Rule "Base Clean Collision StoreL"
Sache(Pcs(Pcm(t,StoreL(v),a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
  if ( ((pcx != pcy) && (cmx == cmy)) && (a1 != a) )
    where Cell(a1,-,Clean()) = data[hash(a)]
==>Sache(Pcs(-,pcx,pcy),cps,mcs,Cms(Cmm(CacheReq(a),cmz,cmy),
    data[hash(a)]:=Cell(a1,-,CachePending())))
    where cmz = switch(cmx)
      case 0: 1
      case 1: 0

Rule "Base Dirty Collision StoreL"
Sache(Pcs(Pcm(t,StoreL(v),a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
  if ( ((pcx != pcy) && (cmx == cmy)) && (a1 != a) )
    where Cell(a1,v,Dirty()) = data[hash(a)]
==>Sache(Pcs(-,pcx,pcy),cps,mcs,Cms(Cmm(Wb(v),a1),cmz,cmy),
    data[hash(a)]:=Cell(a1,v,WbPending())))
    where cmz = switch(cmx)
      case 0: 1
      case 1: 0

Rule "Base Clean or CMiss Commit"
Sache(Pcs(Pcm(t,Commit(a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
  if ( ((pcx != pcy) && (cmx == cpy)) &&
    ((Cell(a,-,-) != cs) || (Cell(a,-,Clean()) == cs)))
    where cs = data[hash(a)]
==>Sache(Pcs(-,pcx,pcy),Cps(Cpm(t,Ack()),cpz,cpy),mcs,cms,data)
    where cpz = switch(cpx)
      case 0: 1
      case 1: 0

Rule "Base Dirty Commit"
Sache(Pcs(Pcm(t,Commit(a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
  if ( ((pcx != pcy) && (cmx == cmy))
    where Cell(a,v,Dirty()) = data[hash(a)]
==>Sache(Pcs(-,pcx,pcy),cps,mcs,Cms(Cmm(Wb(v),a),cmz,cmy),
    data[hash(a)]:=Cell(a,v,WbPending())))
    where cmz = switch(cmx)
      case 0: 1
      case 1: 0
Rule "Base Clean Reconcile"
Sache(Pcs(Pcm(t,Reconcile(),a),pcx,pcy),Cps(-,cpx,cpy),mcs,cms,data)
  if((pcx != pcy) & (cpx == cpy))
    where Cell(a,-,Clean()) = data[hash(a)]
  ==>Sache(Pcs(-,pcx,pcx),Cps(Cpm(t,Ack(),cpz,cpy),mcs,cms,
      data[hash(a):=Invalid()])
    where cpz = switch(cpx)
      case 0: 1
      case 1: 0

Rule "Base Dirty or CMiss Reconcile"
Sache(Pcs(Pcm(t,Reconcile(),a),pcx,pcy),Cps(-,cpx,cpy),mcs,cms,data)
  if((pcx != pcy) & (cpx == cpy) & ((Cell(a,-,-) != cs) | (Cell(a,-,Dirty()) == cs))
    where cs = data[hash(a)]
  ==>Sache(Pcs(-,pcx,pcx),Cps(Cpm(t,Ack(),cpz,cpy),mcs,cms,data)
    where cpz = switch(cpx)
      case 0: 1
      case 1: 0

Rule "Base Mandatory CacheReq Response"
Sache(pcs,cps,Mcs(Mcm(Cache(v),a),mcx,mcy),cms,data)
  if(mcx != mcy)
    where Cell(a,-,CachePending()) = data[hash(a)]
  ==>Sache(pcs,cps,Mcs(-,mcx,mcx),cms,data[hash(a):=Cell(a,v,Clean())])

Rule "Base Mandatory Write Back Done"
Sache(pcs,cps,Mcs(Mcm(WbAcko,a),mcx,mcy),cms,data)
  if(mcx != mcy)
    where Cell(a,v,WbPending()) = data[hash(a)]
  ==>Sache(pcs,cps,Mcs(-,mcx,mcx),cms,data[hash(a):=Cell(a,v,Clean())])

Rule "Base Mandatory Cache Request"
SysMem(Ins(InM(CacheReq(),a,ws),inx,iny),Outs(-,outx,outy),memdata, Nil())
  if(inx != iny) & (outx == outy)
    where MemCell(v,-) = memdata[a]
  ==>SysMem(Ins(-,inx,inx),Outs(OutM(Cache(v),a,ws),outz,outy),memdata, Nil())
    where outz = switch(outx)
      case 0: 1
      case 1: 0

Rule "Base Mandatory Write Back Request"
SysMem(Ins(InM(Wb(v),a,ws),inx,iny),Outs(-,outx,outy),memdata, Nil())
  if(inx != iny) & (outx == outy)
  ==>SysMem(Ins(-,inx,inx),Outs(OutM(WbAcko(),a,ws),outz,outy),
    memdata[a:=MemCell(v,-)],Nil())
    where outz = switch(outx)
      case 0: 1
      case 1: 0
Appendix F: Computer Readable TRS of Cachet-WriterPush

Rule "WriterPush Clean or Dirty LoadL"
Sache(Pcs(Pcm(t,LoadL(a),pcx,pcy),Cps(-,cpx,cpy),mcs,cms,data)
if ( ((pcx != pcy) && (cpx == cpy)) &&
    ((s == Clean()) || (s == Dirty())))
where Cell(a,v,s) = data[hash(a)]
==>Sache(Pes(-,pcx,pcy),Cps(Cpm(t,Value(v)),cpz,cpy),mcs,cms,data)
where cpz = switch(cpx)
case 0: 1
case 1: 0

Rule "WriterPush Clean or Dirty StoreL"
Sache(Pcs(Pcm(t,StoreL(v),a),pcx,pcy),Cps(-,cpx,cpy),mcs,cms,data)
if ( ((pcx != pcy) && (cpx == cpy)) &&
    ((s == Clean()) || (s == Dirty())))
where Cell(a,-,s) = data[hash(a)]
==>Sache(Pes(-,pcx,pcy),Cps(Cpm(t,Ack()),cpz,cpy),mcs,cms,
data[hash(a)]:=Cell(a,v,Dirty))
where cpz = switch(cpx)
case 0: 1
case 1: 0

Rule "WriterPush Invalid LoadL or StoreL"
Sache(Pcs(Pcm(t,i,a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
if ( ((pcx != pcy) && (cmx == cmy)) &&
    ((i == StoreL(-)) || (i == LoadL())))
where InvalidO = data[hash(a)]
==>Sache(Pes(-,pcx,pcy),cps,mcs,Cms(Cmm(CacheReq(),a),cmz,cmy),
data[hash(a):=Cell(a,-,CachePending())])
where cmz = switch(cmx)
case 0: 1
case 1: 0

Rule "WriterPush Clean Collision LoadL or StoreL"
Sache(Pcs(Pcm(t,i,a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
if ( ((pcx != pcy) && (cmx == cmy)) && (a1 != a)) &&
    ((i == StoreL(-)) || (i == LoadL())))
where Cell(a1,-,Clean()) = data[hash(a)]
==>Sache(Pes(-,pcx,pcy),cps,mcs,Cms(Cmm(Purged(),a1),cmz,cmy),
data[hash(a):=Invalid()])
where cmz = switch(cmx)
case 0: 1
case 1: 0
Rule "WriterPush Dirty Collision LoadL or StoreL"
Sache(Pcs(Pcm(t,i,a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
    if ((pcx != pcy) && (cmx == cmy)) && (a1 != a) &&
        (i == StoreL(-)) || (i == LoadL(0))
    where Cell(a1,v,Dirty()) = data[hash(a)]
==>Sache(Pcs(-,pcx,pcy),cps,mcs,Cms(Cmm(Wb(v),a1),cmz,cmy),
data[hash(a):=Cell(a1,v,WbPending())])
    where cmz = switch(cmz)
        case 0: 1
        case 1: 0

Rule "WriterPush Clean or CMiss Commit"
Sache(Pcs(Pcm(t,Commit(),a),pcx,pcy),Cps(-,cpx,cpy),mcs,cms,data)
    if ((pcx != pcy) && (cpx == cpy)) &&
        ((Cell(a,-,-) != cs) || (Cell(a,-,Clean()) == cs))
    where cs = data[hash(a)]
==>Sache(Pcs(-,pcx,cpy),Cps(Cpm(t,Acko),cpz,cpy),mcs,cms,data)
    where cpz = switch(cpx)
        case 0: 1
        case 1: 0

Rule "WriterPush Dirty Commit"
Sache(Pcs(Pcm(t,Commit(),a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
    if ((pcx != pcy) && (cmx == cmy))
    where Cell(a,v,Dirty()) = data[hash(a)]
==>Sache(Pcs(-,pcx,pcy),cps,mcs,Cms(Cmm(Wb(v),a),cmz,cmy),
data[hash(a):=Cell(a,v,WbPending())])
    where cmz = switch(cmz)
        case 0: 1
        case 1: 0

Rule "WriterPush Clean or Dirty or CMiss Reconcile"
Sache(Pcs(Pcm(t,Reconcileo,a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
    if ((pcx != pcy) && (cpx == cpy)) &&
        ((Cell(a,-,-) != cs) ||
         (Cell(a,-,Dirty()) == cs) ||
         (Cell(a,-,Clean()) == cs))
    where cs = data[hash(a)]
==>Sache(Pcs(-,pcx,cpy),Cps(Cpm(t,Acko),cpz,cpy),mcs,cms,data)
    where cpz = switch(cpx)
        case 0: 1
        case 1: 0

Rule "WriterPush Mandatory CacheReq Response"
Sache(pcs,cps,Mcs(Mcm(Cache(v),a),mcx,mcy),cms,data)
    if (mcx != mcy)
    where Cell(a,-,CachePending()) = data[hash(a)]
==>Sache(pcs,cps,Mcs(-,mcx,mcy),cms,data[hash(a):=Cell(a,v,Clean())])
Rule "WriterPush Retry CacheReq"
Sache(pcs,cps,Mcs(Mcm(Retry(),a),mcx,mcy),Cms(-,cmx,cmy),data)
if ((mcx != mcy) && (cmx == cmy))
    where Cell(a,-,CachePending()) = data[hash(a)]
==>Sache(pcs,cps,Mcs(-,mcx,mcx),Cms(Cmm(CacheReq(),a),cmz,cmy),data)
    where cmz = switch(cmx)
        case 0: 1
        case 1: 0

Rule "WriterPush Mandatory Write Back Done"
Sache(pcs,cps,Mcs(Mcm(WbAcko(),a),mcx,mcy),cms(data)
if (mcx != mcy)
    where Cell(a,v,WbPending()) = data[hash(a)]
==>Sache(pcs,cps,Mcs(-,mcx,mcx),cms(data[hash(a)]:=Cell(a,v,Clean())))

Rule "WriterPush Mandatory Write Back Flush"
Sache(pcs,cps,Mcs(Mcm(WbAckFlush(),a),mcx,mcy),cms(data)
if (mcx != mcy)
    where Cell(a,v,WbPending()) = data[hash(a)]
==>Sache(pcs,cps,Mcs(-,mcx,mcx),cms(data[hash(a)]:=Invalid()))

Rule "WriterPush Mandatory Clean Purge Request"
Sache(pcs,cps,Mcs(Mcm(PurgeReq(),a),mcx,mcy),Cms(-,cmx,cmy),data)
if ((mcx != mcy) && (cmx == cmy))
    where Cell(a,-,Clean()) = data[hash(a)]
==>Sache(pcs,cps,Mcs(-,mcx,mcx),Cms(Cmm(Purged(),a),cmz,cmy),
    data[hash(a)]:=Invalid())
    where cmz = switch(cmx)
        case 0: 1
        case 1: 0

Rule "WriterPush Mandatory Dirty Purge Request"
Sache(pcs,cps,Mcs(Mcm(PurgeReq(),a),mcx,mcy),Cms(-,cmx,cmy),data)
if ((mcx != mcy) && (cmx == cmy))
    where Cell(a,v,Dirty()) = data[hash(a)]
==>Sache(pcs,cps,Mcs(-,mcx,mcx),Cms(Cmm(Wb(v),a),cmz,cmy),
    data[hash(a)]:=Cell(a,v,WbPending()))
    where cmz = switch(cmx)
        case 0: 1
        case 1: 0

Rule "WriterPush Mandatory CMiss or WbPending Purge Request"
Sache(pcs,cps,Mcs(Mcm(PurgeReq(),a),mcx,mcy),cms(data)
if (mcx != mcy) && ((Cell(a,-,WbPending()) == cs) || (Cell(a,-,WbPending()) == cs))
    where cs = data[hash(a)]
==>Sache(pcs,cps,Mcs(-,mcx,mcx),cms(data)
Rule "WriterPush Mandatory Cache Request"
SysMem(Ins(InM(CacheReq(a,ws),inx,iny),Outs(-,outx,outy),memdata, Nil())
  if ((inx != iny) && (outx == outy)) && (wsb == 0)
  where MemCell(v,Stable(s0,s1,s2,s3)) = memdata[a]
  wsb = (ws == 0)?s0:(ws == 1)?s1:(ws == 2)?s2:s3
  ==>SysMem(Ins(-,inx,inx),Outs(OutM(Cache(v),a,ws),outz,outy),
  memdata[a:=MemCell(v,Stable(ws == 0)?1:s0,(ws == 1)?1:s1,
  (ws == 2)?1:s2,(ws == 3)?1:s3)],Nil())
  where outz = (outx == 0)?1:0

Rule "WriterPush Unneeded Degenerate Cache Request"
SysMem(Ins(InM(CacheReq(a,ws),inx,iny),outs,memdata, Nil())
  if (inx != iny) && (wsb == 1)
  where MemCell(v,s) = memdata[a]
  Stable(s0,s1,s2,s3) = s
  wsb = (ws == 0)?s0:(ws == 1)?s1:(ws == 2)?s2:s3
  ==>SysMem(Ins(-,inx,inx),outs,memdata, Nil())

Rule "WriterPush Mandatory Write Back"
SysMem(Ins(InM(Wb(v),a,ws),inx,iny),Outs(-,outx,outy),memdata, Nil())
  if ((inx != iny) && (outx == outy)) && ((s0+s1+s2+s3 > 1) && (wsb == 1))
  where MemCell(-,Stable(s0,s1,s2,s3)) = memdata[a]
  wsb = (ws == 0)?s0:(ws == 1)?s1:(ws == 2)?s2:s3
  ==>SysMem(Ins(-,inx,inx),Outs(OutM(PurgeReq(a,ws),a,ws == 0)?1:0),outz,outy),
  memdata[a:=MemCell(-,Transient(s0,s1,s2,s3,v,ws))],
  Purgings(a,ws))
  where outz = (outx == 0)?1:0

Rule "WriterPush Purgings temp state"
SysMem(ins,Outs(-,outx,outy),memdata,Purgings(a,ws))
  if (outx == outy)
  ==>SysMem(ins,Outs(OutM(PurgeReq(a,ws),a,ws == 0)?2:(ws == 1)?2:1),outz,outy),
  memdata,Purgings(a,ws))
  where outz = (outx == 0)?1:0

Rule "WriterPush Purging temp state"
SysMem(Ins(-,inx,-),Outs(-,outx,outy),memdata,Purging(a,ws))
  if (outx == outy)
  ==>SysMem(Ins(-,inx,inx),Outs(OutM(PurgeReq(a,ws),a,ws == 3)?2:3),outz,outy),
  memdata, Nil())
  where outz = (outx == 0)?1:0

Rule "WriterPush Mandatory Fast Write Back"
SysMem(Ins(InM(Wb(v),a,ws),inx,iny),Outs(-,outx,outy),memdata, Nil())
  if ((inx != iny) && (outx == outy)) && ((s0+s1+s2+s3 == 1) && (wsb == 1))
  where MemCell(-,Stable(s0,s1,s2,s3)) = memdata[a]
  wsb = (ws == 0)?s0:(ws == 1)?s1:(ws == 2)?s2:s3
  ==>SysMem(Ins(-,inx,inx),Outs(OutM(WbAck(a,ws),a,ws),outz,outy),
  memdata[a:=MemCell(v,Stable(s0,s1,s2,s3))],Nil())
  where outz = (outx == 0)?1:0
Rule "WriterPush Mandatory Write Back while Transient"

SysMem(Ins(InM(Wb(-),a,ws),inx,iny),Outs(-,outx,outy),memdata, Nil())
if ( ((inx != iny) && (outx == outy)) &&
    ((s0+s1+s2+s3 > 1) && (wsb == 1)))
where MemCell(-,s) = memdata[a]

Transmit(s0,s1,s2,s3,v,ss) = s
wsb = (ws==0)?s0:(ws==1)?s1:(ws==2)?s2:s3

=>SysMem(Ins(-,inx,inx),Outs(OutM(WbAckFlush(),a,ws),outz,outy),
memdata[a:=MemCell(-,Transmit((ws==0)?0:s0,(ws==1)?0:s1,
  (ws==2)?0:s2,(ws==3)?0:s3,v,ss))],Nil())
where outz=(outx==0)?1:0

Rule "WriterPush Flush & WbAck"

SysMem(Ins(InM(Wb(-),a,ws),inx,iny),Outs(-,outx,outy),memdata, Nil())
if ( ((inx != iny) && (outx == outy)) &&
    ((s0+s1+s2+s3 == 1) && (wsb == 1)))
where MemCell(-,Transmit(s0,s1,s2,s3,v,ns)) = memdata[a]

wsb = (ws==0)?s0:(ws==1)?s1:(ws==2)?s2:s3

=>SysMem(Ins(-,inx,inx),Outs(OutM(WbAckFlush(),a,ws),outz,outy),
memdata[a:=MemCell(v,Stable((ns==0)?1:0,(ns==1)?1:0,
  (ns==2)?1:0,(ns==3)?1:0)),WbAcking(a,ns)]
where outz=(outx==0)?1:0

Rule "WriterPush WbAcking temp state"

SysMem(Ins(-,inx,inx),Outs(-,outx,outy),memdata,WbAcking(a,ns))
if (outx == outy)

=>SysMem(Ins(-,inx,inx),Outs(OutM(WbAck(),a,ns),outz,outy),memdata, Nil())
where outz=(outx==0)?1:0

Rule "WriterPush Mandatory Purged while Stable"

SysMem(Ins(InM(Purgedo,a,ws),inx,iny),outs,memdata, Nil())
if ( (inx != iny) && (wsb == 1) )
where MemCell(v,Stable(s0,s1,s2)) = memdata[a]

wsb = (ws==0)?s0:(ws==1)?s1:(ws==2)?s2:s3

=>SysMem(Ins(-,inx,inx),outs,memdata[a:=MemCell(v,
  Stable((ws==0)?0:s0,(ws==1)?0:s1,(ws==2)?0:s2,(ws==3)?0:s3))],Nil())

Rule "WriterPush Mandatory Purged while Transient"

SysMem(Ins(InM(Purgedo,a,ws),inx,iny),outs,memdata, Nil())
if ( (inx != iny) && ((s0+s1+s2+s3 > 1) && (wsb == 1)) )
where MemCell(-,Transmit(s0,s1,s2,s3,v,ss)) = memdata[a]

wsb = (ws==0)?s0:(ws==1)?s1:(ws==2)?s2:s3

=>SysMem(Ins(-,inx,inx),outs,memdata[a:=MemCell(v,Transmit((ws==0)?0:s0,
  (ws==1)?0:s1,(ws==2)?0:s2,(ws==3)?0:s3,v,ss))],Nil())
Rule "WriterPush Mandatory Writeback Completes"
SysMem(Ins(InM(Purged()),a,ws),inx,iny), Outs(,outx, outy), memdata, Nil())
if ((inx != iny) && (outx == outy)) &&
((s0+s1+s2+s3 == 1) && (wsb == 1))
where MemCell(v,s) = memdata[a]
  Transient(s0,s1,s2,s3,v,ns) = s
  wsb = (ws==0)?s0:(ws==1)?s1:(ws==2)?s2:s3
==>SysMem(Ins(,inx,inx), Outs(OutM(WbAck(),a, ns),outz, outy),
  memdata[a]:=MemCell(v,Stable(ns==0)?1:0,(ns==1)?1:0,
  (ns==2)?1:0,(ns==3)?1:0))],Nil())
where outz=(outx==0)?1:0
Appendix G: Computer Readable TRS of Cachet-Migratory

Rule "Migratory Clean or Dirty LoadL"
Sache(Pcs(Pcm(t,LoadL(a),pcx,pcy),Cps(-,cpx,cpy),mcs,cms,data)
  if ( ((pcx != pcy) && (cpx == cpy) ) &&
     ((s == Clean()) || (s == Dirty())) )
  where Cell(a,v,s) = data[hash(a)]
==>Sache(Pcs(-,pcx,cpy),Cps(Cpm(t,Value(v)),cpz,cpy),mcs,cms,data)
  where cpz = switch(cpx)
    case 0: 1
    case 1: 0

Rule "Migratory Clean or Dirty StoreL"
Sache(Pcs(Pcm(t,StoreL(v),a),pcx,pcy),Cps(-,cpx,cpy),mcs,cms,data)
  if ( ((pcx != pcy) && (cpx == cpy) ) &&
     ((s == Clean()) || (s == Dirty()) )
  where Cell(a,-,s) = data[hash(a)]
==>Sache(Pcs(-,pcx,cpy),Cps(Cpm(t,Ack()),cpz,cpy),mcs,cms,
     data[hash(a):=Cell(a,v,Dirty())])
  where cpz = switch(cpx)
    case 0: 1
    case 1: 0

Rule "Migratory Invalid LoadL or StoreL"
Sache(Pcs(Pcm(t,i,a),pcx,pcy),Cps(-,cpx,cpy),CMS(-,cmx,cmy),data)
  if ( ((pcx != pcy) && (cmx == cmy) ) &&
     ((i == StoreL(-)) || (i == LoadL()) )
  where Invalid() = data[hash(a)]
==>Sache(Pcs(-,pcx,cpy),Cps(Cmm(CacheReq,a),cmz,cmy),
     data[hash(a):=Cell(a,-,CachePending())])
  where cmz = switch(cmx)
    case 0: 1
    case 1: 0

Rule "Migratory Clean Collision LoadL or StoreL"
Sache(Pcs(Pcm(t,i,a),pcx,pcy),Cps(-,cpx,cpy),CMS(-,cmx,cmy),data)
  if ( ((pcx != pcy) && (cmx == cmy) ) && (a != a) ) &&
     ((i == StoreL(-)) || (i == LoadL()) )
  where Cell(a1,-,Clean()) = data[hash(a)]
==>Sache(Pcs(-,pcx,cpy),Cps(mcs,Cms(Cmm(Purged(),a1),cmz,cmy),
     data[hash(a):=Invalid()]])
  where cmz = switch(cmx)
    case 0: 1
    case 1: 0
Rule "Migratory Dirty Collision LoadL or StoreL"
Sache(Pcs(Pcm(t,i,a),pcx,pcy),cps,mcs,Cms(-,cmx,cmy),data)
    if ( ((pcx != pcy) && (cmx == cmy)) && (a1 != a)) &&
        ( (i == StoreL(->)) | (i == LoadL()) )
    where Cell(a1,v,Dirty()) = data[hash(a)]
===>Sache(Pcs(-,pcx,pcy),cps,mcs,Cms(Cmm(Wb(v),a1),cmz,cmy),
        data[hash(a):=Invalid()])
    where cmz = switch(cmx)
        case 0: 1
        case 1: 0

Rule "Migratory Clean or Dirty or CMiss Commit"
Sache(Pcs(Pcm(t,Commit(),a),pcx,pcy),Cps(-,cpx,cpy),mcs,cms,data)
    if ( ((pcx!=pcy) && (cpx == cpy)) &&
        ((Cell(a,-,-) != cs) | |
        (Cell(a,-,Dirty()) == cs) | |
        (Cell(a,-,Clean()) == cs) )
    where cs = data[hash(a)]
===>Sache(Pcs(-,pcx,pcx),Cps(Cpm(t,Ack()),cpz,cpy),mcs,cms,data)
    where cpz = switch(cpx)
        case 0: 1
        case 1: 0

Rule "Migratory Clean or Dirty or CMiss Reconcile"
Sache(Pcs(Pcm(t,Reconcileo,a),pcx,pcy),Cps(-,cpx,cpy),mcs,cms,data)
    if ( ((pcx!=pcy) && (cpx == cpy)) &&
        ((Cell(a,-,-) != cs) | |
        (Cell(a,-,Dirty()) == cs) | |
        (Cell(a,-,Cleano) == cs) )
    where cs = data[hash(a)]
===>Sache(Pcs(-,pcx,pcx),Cps(Cpm(t,Acko),cpz,cpy),mcs,cms,data)
    where cpz = switch(cpx)
        case 0: 1
        case 1: 0

Rule "Migratory Mandatory CacheReq Response"
Sache(pcs,cps,Mcs(Mcm(Cache(v),a),mcx,mcy),cms,data)
    if (mcx != mcy)
    where Cell(a,-,CachePending()) = data[hash(a)]
===>Sache(pcs,cps,Mcs(-,mcx,mcx),cms,data[hash(a):=Cell(a,v,Cleano)])

Rule "Migratory Retry CacheReq"
Sache(pcs,cps,Mcs(Mcm(Retryo,a),mcx,mcy),Cms(-,cmx,cmy),data)
    if ((mcx != mcy) && (cmx == cmy))
    where Cell(a,-,CachePending()) = data[hash(a)]
===>Sache(pcs,cps,Mcs(-,mcx,mcx),Cms(Cmm(CacheReq(),a),cmz,cmy),data)
    where cmz = switch(cmz)
        case 0: 1
        case 1: 0
Rule "Migratory Mandatory Clean Purge Request"
Sache(pcs,cps,Mcs(Mcm(PurgeReq(),a),mcx,mcy),Cms(-,cmx,cmy),data)
  if ((mcx != mcy) && (cmx == cmy))
    where Cell(a,-,Clean()) = data[\text{hash}(a)]
  ==> Sache(pcs,cps,Mcs(-,mcx,mcx),Cms(Cmm(Purged(),a),cmz,cmy),
         data[\text{hash}(a):=\text{Invalid}]())
    where cmz = \text{switch}(cmx)
      case 0: 1
      case 1: 0

Rule "Migratory Mandatory Dirty Purge Request"
Sache(pcs,cps,Mcs(Mcm(PurgeReq(),a),mcx,mcy),Cms(-,cmx,cmy),data)
  if ((mcx != mcy) && (cmx == cmy))
    where Cell(a,v,Dirty()) = data[\text{hash}(a)]
  ==> Sache(pcs,cps,Mcs(-,mcx,mcx),Cms(Cmm(Wb(v),a),cmz,cmy),
            data[\text{hash}(a):=\text{Invalid}]())
    where cmz = \text{switch}(cmx)
      case 0: 1
      case 1: 0

Rule "Migratory Mandatory CMiss Purge Request"
Sache(pcs,cps,Mcs(Mcm(PurgeReq(),a),mcx,mcy),cms,data)
  if (mcx != mcy) && (Cell(a,-,-) != cs)
    where cs = data[\text{hash}(a)]
  ==> Sache(pcs,cps,Mcs(-,mcx,mcx),cms,data)

Rule "Migratory Mandatory Cache Request"
SysMem(Ins(InM(CacheReq(),a,ws),inx,iny),Outs(-,outx,outy),memdata, Nil())
  if (inx != iny) && (outx == outy)
    where MemCell(v,Noone()) = memdata[a]
  ==> SysMem(Ins(-,inx,inx),Outs(OutM(Cache(v),a,ws),outz,outy),
                memdata[a:=MemCell(v,\text{Exclusive}(ws))],Nil())
    where outz = \text{switch}(outx)
      case 0: 1
      case 1: 0

Rule "Migratory Mandatory CacheReq Bump Out"
SysMem(Ins(InM(CacheReq(),a,ws),inx,iny),Outs(-,outx,outy),memdata, Nil())
  if (inx != iny) && (outx == outy)
    where MemCell(v,\text{Exclusive}(holder)) = memdata[a]
  ==> SysMem(Ins(-,inx,inx),Outs(OutM(PurgeReq(),a,holder),outz,outy),
                memdata[a:=MemCell(v,\text{MigTrans}(holder, ws))],Nil())
    where outz = \text{switch}(outx)
      case 0: 1
      case 1: 0

Rule "Migratory Cache Request while MigTrans"
SysMem(Ins(InM(CacheReq(),a,ws),inx,iny),Outs(-,outx,outy),memdata, Nil())
  if (inx != iny) && (outx == outy)
    where MemCell(v,\text{Transient(-,-,-,-,-,-)}) = memdata[a]
  ==> SysMem(Ins(-,inx,inx),Outs(OutM(Retry(),a,ws),not(outx),outy),
             memdata,Nil())
Rule "Migratory Mandatory Purged while Exclusive"
SysMem(Ins(InM(Purged(),a,ws),inx,iny),outs,memdata, Nil())
  if (inx != iny)
    where MemCell(v,Exclusive(ws)) = memdata[a]
  ==> SysMem(Ins(-,inx,inx),outs,memdata[a:=MemCell(v,Noone())],Nil())

Rule "Migratory Mandatory Purged while MigTrans"
SysMem(Ins(InM(Purged(),a,ws),inx,iny),Outs(-,outx,outy),memdata, Nil())
  if ((inx != iny) & & (outx == outy))
    where MemCell(v,MigTrans(ws,ns)) = memdata[a]
  ==> SysMem(Ins(-,inx,inx),Outs(OutM(Cache(v),a,ns),outz,outy),
    memdata[a:=MemCell(v,Exclusive(ns))],Nil())
    where outz = switch(outx)
      case 0: 1
      case 1: 0

Rule "Migratory Mandatory Write Back while Exclusive"
SysMem(Ins(InM(Wb(a,ws),a,ws),inx,iny),outs,memdata, Nil())
  if (inx != iny)
    where MemCell(-,Exclusive(ws)) = memdata[a]
  ==> SysMem(Ins(-,inx,inx),outs,memdata[a:=MemCell(v,Noone())],Nil())

Rule "Migratory Mandatory Write Back while MigTrans"
SysMem(Ins(InM(Wb(v),a,ws),inx,iny),Outs(-,outx,outy),memdata, Nil())
  if ((inx != iny) & & (outx == outy))
    where MemCell(-,MigTrans(ws,ns)) = memdata[a]
  ==> SysMem(Ins(-,inx,inx),Outs(OutM(Cache(v),a,ns),outz,outy),
    memdata[a:=MemCell(v,Exclusive(ns))],Nil())
    where outz = switch(outx)
      case 0: 1
      case 1: 0
Bibliography


