Design and Verification of Adaptive Cache Coherence Protocols

by

Xiaowei Shen

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of
the Requirements for the Degree of
Doctor of Philosophy

at the

Massachusetts Institute of Technology

February, 2000

Signature of Author______________________________
Department of Electrical Engineering and Computer Science
January 5, 2000

Certified by______________________________
Arvind
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by______________________________
Arthur C. Smith
Chairman, Committee on Graduate Students

Design and Verification of Adaptive Cache Coherence Protocols

by

Xiaowei Shen

Submitted to the Department of Electrical Engineering and Computer Science
on January 5, 2000
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

We propose to apply Term Rewriting Systems (TRSs) to modeling computer architectures and distributed protocols. TRSs offer a convenient way to precisely describe asynchronous systems and can be used to verify the correctness of an implementation with respect to a specification. This dissertation illustrates the use of TRSs by giving the operational semantics of a simple instruction set, and a processor that implements the same instruction set on a micro-architecture that allows register renaming and speculative execution.

A mechanism-oriented memory model called Commit-Reconcile & Fences (CRF) is presented that allows scalable implementations of shared memory systems. The CRF model exposes a semantic notion of caches, referred to as saches, and decomposes memory access operations into simpler instructions. In CRF, a memory load operation becomes a Reconcile followed by a Loadl, and a memory store operation becomes a Storel followed by a Commit. The CRF model can serve as a stable interface between computer architects and compiler writers.

We design a family of cache coherence protocols for distributed shared memory systems. Each protocol is optimized for some specific access patterns, and contains a set of voluntary rules to provide adaptivity that can be invoked whenever necessary. It is proved that each protocol is a correct implementation of CRF, and thus a correct implementation of any memory model whose programs can be translated into CRF programs. To simplify protocol design and verification, we employ a novel two-stage design methodology called Imperative-&-Directive that addresses the soundness and liveness concerns separately throughout protocol development.

Furthermore, an adaptive cache coherence protocol called Cachet is developed that provides enormous adaptivity for programs with different access patterns. The Cachet protocol is a seamless integration of multiple micro-protocols, and embodies both intra-protocol and inter-protocol adaptivity that can be exploited via appropriate heuristic mechanisms to achieve optimal performance under changing program behaviors. The Cachet protocol allows store accesses to be performed without the exclusive ownership, which can notably reduce store latency and alleviate cache thrashing due to false sharing.

Thesis Supervisor: Arvind

Title: Professor of Electrical Engineering and Computer Science

Acknowledgment

I would like to thank Arvind, my advisor, for his continuing guidance and support during my six years of fruitful research. His sharp sense of research direction, great enthusiasm and strong belief in the potential of this research has been a tremendous driving force for the completion of this work. The rapport between us makes research such an exciting experience that our collaboration finally produces something that we are both proud of.

This dissertation would not have been possible without the assistance of many people. I am greatly indebted to Larry Rudolph, who has made invaluable suggestions in many aspects of my work, including some previous research that is not included in this dissertation. I have always enjoyed bringing vague and sometimes bizarre ideas to him; our discussions in offices, lounges and corridors have led to some novel results.

During my final years at MIT, I have had the pleasure of working with Joe Stoy from Oxford University. It is from him that I learned more about language semantics and formal methods, and with his tremendous effort some of the protocols I designed were verified using theorem provers. Many faculty members at MIT, especially Martin Rinard and Charles Leiserson, have provided interesting comments that helped improve the work. I would also like to thank Guang Gao at University of Dalaware for his useful feedbacks. My thanks also go to our friends at IBM Thomas J. Watson Research Center: Basil, Chuck, Eknath, Marc and Vivek.

I want to thank my colleagues at the Computation Structures Group at the MIT Laboratory for Computer Science. My motivation of this research originates partly from those exciting discussions with Boon Ang and Derek Chiou about the Start-NG and Start-Voyager projects. Jan-Willem Maessen explained to me the subtle issues about the Java memory model. James Hoe helped me in many different ways, from showing me around the Boston area to bringing me to decent Chinese restaurants nearby. Andy Shaw provided me with great career advice during our late night talks. I would also like to say "thank you" to Andy Boughton, R. Paul Johnson, Alejandro Caro, Daniel Rosenband and Keith Randall. There are many other people whose names are not mentioned here, but this does not mean that I have forgot you or your help. It is a privilege to have worked with so many bright and energetic people. Your talent and friendship have made MIT such a great place to live (which should also be blamed for my lack of incentive to graduate earlier).

I am truly grateful to my parents for giving me all the opportunities in the world to explore my potentials and pursue my dreams. I would also like to thank my brothers for their everlasting encouragement and support. I owe my deepest gratitude to my wife Yue for her infinite patience that accompanied me along this long journey to the end and for her constant love that pulled me through many difficult times. Whatever I have, I owe to her.

Xiaowei Shen
Massachusetts Institute of Technology
Cambridge, Massachusetts
January, 2000

Contents

1 Introduction **13**

1.1 Memory Models 14

1.1.1 Sequential Consistency 15

1.1.2 Memory Models of Modern Microprocessors 16

1.1.3 Architecture-Oriented Memory Models 17

1.1.4 Program-Oriented Memory Models 18

1.2 Cache Coherence Protocols 20

1.2.1 Snoopy Protocols and Directory-based Protocols 21

1.2.2 Adaptive Cache Coherence Protocols 21

1.2.3 Verification of Cache Coherence Protocols 22

1.3 Contributions of the Thesis 23

2 Using TRS to Model Architectures **26**

2.1 Term Rewriting Systems 26

2.2 The AX Instruction Set 27

2.3 Register Renaming and Speculative Execution 30

2.4 The P_S Speculative Processor 32

2.4.1 Instruction Fetch Rules 33

2.4.2 Arithmetic Operation and Value Propagation Rules 34

2.4.3 Branch Completion Rules 35

2.4.4 Memory Access Rules 36

2.5 Using TRS to Prove Correctness 37

2.5.1 Verification of Soundness 38

2.5.2 Verification of Liveness 40

2.6 The Correctness of the P_S Model 43

2.7 Relaxed Memory Access Operations 45

3 The Commit-Reconcile & Fences Memory Model **49**

3.1 The CRF Model 49

3.1.1 The CR Model 50

3.1.2 Reordering and Fence Rules 53

3.2 Some Derived Rules of CRF 55

3.2.1 Stalled Instruction Rules 55

3.2.2 Relaxed Execution Rules 56

3.3 Coarse-grain CRF Instructions 57

3.4 Universality of the CRF Model 59

3.5 The Generalized CRF Model 61

4 The Base Cache Coherence Protocol 65

4.1 The Imperative-&-Directive Design Methodology 65

4.2 The Message Passing Rules 67

4.3 The Imperative Rules of the Base Protocol 71

4.4 The Base Protocol 74

4.5 Soundness Proof of the Base Protocol 76

4.5.1 Some Invariants of Base 76

4.5.2 Mapping from Base to CRF 77

4.5.3 Simulation of Base in CRF 79

4.5.4 Soundness of Base 82

4.6 Liveness Proof of the Base Protocol 82

4.6.1 Some Invariants of Base 82

4.6.2 Liveness of Base 84

5 The Writer-Push Cache Coherence Protocol 86

5.1 The System Configuration of the WP Protocol 87

5.2 The Imperative Rules of the WP Protocol 88

5.3 The WP Protocol 90

5.3.1 Mandatory Rules 91

5.3.2 Voluntary Rules 94

5.3.3 FIFO Message Passing 95

5.3.4 Potential Optimizations 96

5.4 Soundness Proof of the WP Protocol 97

5.4.1 Some Invariants of WP 97

5.4.2 Mapping from WP to CRF 98

5.4.3 Simulation of WP in CRF 101

5.4.4 Soundness of WP 104

5.5 Liveness Proof of the WP Protocol 104

5.5.1 Some Invariants of WP 106

5.5.2 Liveness of WP 111

5.6 An Update Protocol from WP 113

5.7 An Alternative Writer-Push Protocol 114

6 The Migratory Cache Coherence Protocol 115

6.1 The System Configuration of the Migratory Protocol 115

6.2 The Imperative Rules of the Migratory Protocol 116

6.3 The Migratory Protocol 118

6.4 Soundness Proof of the Migratory Protocol 122

6.4.1 Some Invariants of Migratory 122

6.4.2 Mapping from Migratory to CRF 122

6.4.3 Simulation of Migratory in CRF 123

6.4.4 Soundness of Migratory 125

6.5 Liveness Proof of the Migratory Protocol 126

6.5.1 Some Invariants of Migratory 126

6.5.2 Liveness of Migratory 131

7 Cachet: A Seamless Integration of Multiple Micro-protocols 133
7.1 Integration of Micro-protocols . . . 133
7.1.1 Putting Things Together . . . 135
7.1.2 Dynamic Micro-protocol Switch . . . 135
7.2 The System Configuration of the Cachet Protocol . . . 137
7.2.1 Cache and Memory States . . . 137
7.2.2 Basic and Composite Messages . . . 138
7.3 The Imperative Rules of the Cachet Protocol . . . 140
7.3.1 Imperative Processor Rules . . . 140
7.3.2 Imperative Cache Engine Rules . . . 140
7.3.3 Imperative Memory Engine Rules . . . 142
7.3.4 Composite Imperative Rules . . . 145
7.4 The Cachet Cache Coherence Protocol . . . 147
7.4.1 Processor Rules of Cachet . . . 147
7.4.2 Cache Engine Rules of Cachet . . . 147
7.4.3 Memory Engine Rules of Cachet . . . 149
7.4.4 Derivation of Cachet from Imperative and Directive Rules . . . 153
7.5 The Composite Rules of Cachet . . . 154

8 Conclusions 161
8.1 Future Work . . . 164

A The Cachet Protocol Specification 166

List of Figures

1.1 Impact of Architectural Optimizations on Program Behaviors 17

2.1 AX: A Minimalist RISC Instruction Set 28
2.2 P_B: A Single-Cycle In-Order Processor 29
2.3 Operational Semantics of AX 30
2.4 P_S: A Processor with Register Renaming and Speculative Execution 31
2.5 P_S Instruction Fetch Rules 34
2.6 P_S Arithmetic Operation and Value Propagation Rules 35
2.7 P_S Branch Completion Rules 36
2.8 P_S Memory Instruction Dispatch and Completion Rules 36
2.9 A Simple Processor-Memory Interface 37
2.10 Forward Draining 38
2.11 Backward Draining 39
2.12 Combination of Forward and Backward Draining 40
2.13 Draining the Processor 43
2.14 Simulation of Instruction Fetch Rules 44

3.1 CRF Instructions 50
3.2 System Configuration of CRF 50
3.3 Producer-Consumer Synchronization in CRF 51
3.4 Summary of CR Rules 52
3.5 Semantic Cache State Transitions of CRF 53
3.6 Instruction Reordering Table of CRF 54
3.7 System Configuration of GCRF 63
3.8 Summary of GCRF Rules (except reordering rules) 64

4.1 The Imperative-&-Directive Design Methodology 66
4.2 System Configuration of Base 68
4.3 Cache State Transitions of Base 72
4.4 Imperative Rules of Base 73
4.5 The Base Protocol 74
4.6 Simulation of Base in CRF 79
4.7 Derivation of Base from Imperative Rules 81
4.8 Simulation of CRF in Base 82

5.1 System Configuration of WP 87
5.2 Protocol Messages of WP 88
5.3 Cache State Transitions of WP's Imperative Operations 89
5.4 Imperative Rules of WP 90

5.5 The WP Protocol 92
5.6 Cache State Transitions of WP 94
5.7 Simplified Memory Engine Rules of WP 96
5.8 Simulation of WP in CRF 103
5.9 Derivation of WP from Imperative & Directive Rules 105
5.10 Memory Engine Rules of an Update Protocol 113
5.11 An Alternative Writer-Push Protocol 114

6.1 System Configuration of Migratory 116
6.2 Cache State Transitions of Migratory's Imperative Operations 117
6.3 Imperative Rules of Migratory 118
6.4 The Migratory Protocol 119
6.5 Cache State Transitions of Migratory 120
6.6 Simulation of Migratory in CRF 125
6.7 Derivation of Migratory from Imperative & Directive Rules 126

7.1 Different Treatment of Commit, Reconcile and Cache Miss 134
7.2 Downgrade and Upgrade Operations 136
7.3 Protocol Messages of Cachet 138
7.4 Composite Messages of Cachet 139
7.5 Imperative Processor Rules of Cachet 141
7.6 Imperative Cache Engine Rules of Cachet 142
7.7 Cache State Transitions of Cachet 143
7.8 Imperative Memory Engine Rules of Cachet 144
7.9 Composite Imperative Rules of Cachet 145
7.10 Simulation of Composite Imperative Rules of Cachet 146
7.11 Processor Rules of Cachet 148
7.12 Cache Engine Rules of Cachet 149
7.13 Memory Engine Rules of Cachet (Rule MM1 is strongly fair) 150
7.14 Derivation of Processor Rules of Cachet 154
7.15 Derivation of Cache Engine Rules of Cachet 155
7.16 Derivation of Memory Engine Rules of Cachet 156
7.17 Composite Rules of Cachet 157
7.18 Simulation of Composite Rules of Cachet 158

A.1 Cachet: The Processor Rules 166
A.2 Cachet: The Cache Engine Rules 167
A.3 Cachet: The Memory Engine Rules 168
A.4 FIFO Message Passing and Buffer Management 169

Chapter 1

Introduction

Shared memory multiprocessor systems provide a global memory image so that processors running parallel programs can exchange information and synchronize with one another by accessing shared variables. In large scale shared memory systems, the physical memory is typically distributed across different sites to achieve better performance. Distributed Shared Memory (DSM) systems implement the shared memory abstraction with a large number of processors connected by a network, combining the scalability of network-based architectures with the convenience of shared memory programming. Caching technique allows shared variables to be replicated in multiple sites simultaneously to reduce memory access latency. DSM systems rely on cache coherence protocols to ensure that each processor can observe the semantic effect of memory access operations performed by another processor in time.

The design of cache coherence protocols plays a crucial role in the construction of shared memory systems because of its profound impact on the overall performance and implementation complexity. It is also one of the most complicated problems because efficient cache coherence protocols usually incorporate various optimizations, especially for cache coherence protocols that implement relaxed memory models. This thesis elaborates on several relevant issues about cache coherence protocols for DSM systems: what memory model should be supported, what adaptivity can be provided and how sophisticated and adaptive cache coherence protocols can be designed and verified.

A shared memory system implements a memory model that defines the semantics of memory access instructions. An ideal memory model should allow efficient and scalable implementations while still have simple semantics for the architect and the compiler writer to reason about. Although various memory models have been proposed, there is little consensus on a memory model that shared memory systems should support. Sequential consistency is easy for programmers to understand and use, but it often prohibits many architectural and compiler optimizations. On the other hand, relaxed memory models may allow various optimizations, but their ever-changing and implementation-dependent definitions have created a situation where even experienced architects may have difficulty articulating precisely the impact of these memory models on program behaviors.

What we are proposing is a mechanism-oriented memory model called Commit-Reconcile & Fences (CRF), which exposes both data replication and instruction reordering at the instruction set architecture level. CRF is intended for architects and compiler writers rather than for high-level parallel programming. One motivation underlying CRF is to eliminate the *modèle de l'année* aspect of many of the existing relaxed memory models while still permit efficient implementations. The CRF model permits aggressive cache coherence protocols because no operation explicitly or implicitly involves more than one semantic cache. A novel feature of CRF is that many memory models can be expressed as restricted versions of CRF in that programs written under those memory models can be translated into efficient CRF programs. Translations of programs written under memory models such as sequential consistency and release consistency into CRF programs are straightforward.

Parallel programs have various memory access patterns. It is highly desirable that a cache coherence protocol can adapt its actions to changing program behaviors. This thesis attacks the adaptivity problem from a new perspective. We develop an adaptive cache coherence protocol called Cachet that provides a wide scope of adaptivity for DSM systems. The Cachet protocol is a seamless integration of several micro-protocols, each of which has been optimized for a particular memory access pattern. Furthermore, the Cachet protocol implements the CRF model, therefore, it is automatically an implementation for all the memory models whose programs can be translated into CRF programs.

Cache coherence protocols can be extremely complicated, especially in the presence of various optimizations. It often takes much more time in verifying the correctness of cache coherence protocols than in designing them, and rigorous reasoning is the only way to avoid subtle errors in sophisticated cache coherence protocols. This is why Term Rewriting Systems (TRSs) is chosen as the underlying formalism to specify and verify computer architectures and distributed protocols. We use TRSs to define the operational semantics of the CRF memory model so that each CRF program has some well-defined operational behavior. The set of rewriting rules can be used by both architects and compiler writers to validate their implementations and optimizations. We can prove the soundness of a cache coherence protocol by showing that the TRS specifying the protocol can be simulated by the TRS specifying the memory model.

The remainder of this chapter gives some background about memory models and cache coherence protocols. In Section 1.1, we give an overview of memory models, from sequential consistency to some relaxed memory models. Section 1.2 discusses cache coherence protocols and some common verification techniques of cache coherence protocols. Section 1.3 is a summary of major contributions of the thesis and outline of the thesis organization.

1.1 Memory Models

Caching and instruction reordering are ubiquitous features of modern computer systems and are necessary to achieve high performance. For uniprocessor systems, these features are mostly transparent and exposed only for low-level memory-mapped input and output operations. For multiprocessor systems, however, these features are anything but transparent. Indeed, a whole area of research has evolved around what view of memory should be presented to the programmer, the compiler writer, and the computer architect.

The essence of memory models is the correspondence between each load instruction and the store instruction that supplies the data retrieved by the load. The memory model of uniprocessor systems is intuitive: a load operation returns the most recent value written to the address, and a store operation binds the value for subsequent load operations. In parallel systems, notions such as "the most recent value" can become ambiguous since multiple processors access memory concurrently. Therefore, it can be difficult to specify the resulting memory model precisely at the architecture level [33, 36, 97]. Surveys of some well-known memory models can be found elsewhere [1, 66].

Memory models can be generally categorized as architecture-oriented and program-oriented. One can think of an architecture-oriented model as the low-level interface between the compiler and the underlying architecture, while a program-oriented model the high-level interface between the compiler and the program. Many architecture-oriented memory models [61, 86, 117, 121] are direct consequences of microarchitecture optimizations such as write-buffers and non-blocking caches. Every programming language has a high-level memory model [19, 56, 54, 92], regardless of whether it is described explicitly or not. The compiler ensures that the semantics of a program is preserved when its compiled version is executed on an architecture with some low-level memory model.

1.1.1 Sequential Consistency

Sequential consistency [72] has been the dominant memory model in parallel computing for decades due to its simplicity. A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. Sequential consistency requires that memory accesses be performed in-order on each processor, and be atomic with respect to each other. This is clearly at odds with both instruction reordering and data caching.

Sequential consistency inevitably prevents many architecture and compiler optimizations. For example, the architect has to be conservative in what can be reordered although dynamic instruction reordering is desirable in the presence of unpredictable memory access latencies. The compiler writer is affected because parallel compilers often use existing sequential compilers as a base, and sequential compilers reorder instructions based on conventional dataflow analysis. Thus, any transformation involving instruction reordering either has to be turned off, or at least requires more sophisticated analysis [70, 109].

The desire to achieve higher performance has led to various relaxed memory models, which can provide more implementation flexibility by exposing optimizing features such as instruction reordering and data caching at the programming level. Relaxed memory models usually weaken either the sequentiality constraint or the atomicity constraint of sequential consistency. A weak-ordering model allows certain memory accesses to be performed out-of-order, while a weak-atomicity model allows a store access to be performed in some non-atomic fashion. A relaxed memory model can accommodate both weak-ordering and weak-atomicity features simultaneously. Note that the distinction between weak-ordering and weak-atomicity is purely based on semantics. For example, a weak-ordering system can employ data caching to improve performance, provided that the existence of caches is semantically transparent.

1.1.2 Memory Models of Modern Microprocessors

Modern microprocessors [62, 86, 117, 121] support relaxed memory models that allow memory accesses to be reordered. For example, the IA-64 architecture [62] allows four memory ordering semantics: unordered, release, acquire or fence. Unordered data accesses may become visible in any order. Release data accesses guarantee that all previous data accesses are made visible prior to being made visible themselves. Acquire data accesses guarantee that they are made visible prior to all subsequent data accesses. Fence operations combine the release and acquire semantics to guarantee that all previous data accesses are made visible prior to any subsequent data accesses being made visible.

Modern microprocessors often provide memory fences that can be used to ensure proper ordering constraints whenever necessary. Examples of memory fences include PowerPC's Sync instruction and Sparc's Membar instruction. Special instructions such as conditional instructions and synchronization instructions can also behave as memory fences in some processors.

Different manufacturers often have different memory models; even the same manufacturer can have different memory models for different generations of microprocessors. In Sparc [121], for example, Total Store Order allows a load instruction to be performed before outstanding store instructions complete, which virtually models FIFO write-buffers. Partial Store Order further allows stores to be reordered so that stores to the same cache line can be merged in write-buffers. Relaxed Memory Order allows loads and stores to be performed in arbitrary order, provided that data dependencies of memory accesses are preserved.

Many architectural optimizations that are transparent in uniprocessors become visible in multiprocessor systems. Different implementations of memory access operations can cause subtle difference in observable behaviors of parallel programs, giving rise to different memory models. Figure 1.1 gives examples that show the impact of architectural optimizations on program behaviors. In all the examples, the initial value of a variable is zero.

Example 1 shows a program that captures the essence of the Dekker's algorithm to enforce mutual exclusion. It is impossible that both registers r_1 and r_2 obtain zero in sequential consistency. With write-buffers, however, both registers r_1 and r_2 may get zero since a load can be performed before the previous store completes. In Example 2, each processor uses an extra load as a memory fence to prevent the following load to bypass the previous store.

Example 1:	Can both registers r_1 and r_2 obtain 0?	
	Processor 1	*Processor 2*
	Store(flag_1,1);	Store(flag_2,1);
	r_1 := Load(flag_2);	r_2 := Load(flag_1);
Example 2:	With write-buffers, can both registers r_1 and r_2 obtain 0?	
	Processor 1	*Processor 2*
	Store(flag_1,1);	Store(flag_2,1);
	r_3 := Load(flag_1);	r_4 := Load(flag_2);
	r_1 := Load(flag_2);	r_2 := Load(flag_1);
Example 3:	Can registers r_1 and r_2 obtain 1 and 0, respectively?	
	Processor 1	*Processor 2*
	Store(*buf*,1);	r_1 := Load(*flag*);
	Store(*flag*,1);	r_2 := Load(*buf*);
Example 4:	Can registers r_1 and r_2 obtain 1 and 0, respectively?	
	Processor 1	*Processor 2*
	Store(*buf*,1);	L: r_1 := Load(*flag*);
	Fence;	Jz(r_1,L);
	Store(*flag*,1);	r_2 := Load(*buf*);

Figure 1.1: Impact of Architectural Optimizations on Program Behaviors

Unfortunately, the extra load instruction would make no semantic difference in the presence of short-circuiting, which allows the load to retrieve the data from the write-buffer that contains an outstanding store to the address. Example 3 shows a program that implements the producer-consumer synchronization. In the presence of Non-FIFO write-buffers or non-blocking caches, it may appear to processor 2 that processor 1 asserts the flag before it writes the new data to the buffer. The question for Example 4 is whether the branch instruction behaves as an implicit memory fence. With speculative execution, processor 2 can speculatively perform the load operation to the buffer before it observes the value of the flag.

As a reaction to ever-changing memory models and their complicated and imprecise definitions, there is a desire to go back to the simple, easy-to-understand sequential consistency, even though there are a plethora of problems in its high-performance implementation and no compiler writer seems to adhere to its semantics. Ingenious solutions have been devised to maintain the sequential consistency semantics so that programmers cannot detect if and when the memory accesses are out-of-order or non-atomic. Recent advances in speculative execution permit reordering of memory accesses without affecting the sequentiality of sequential consistency [48, 51, 125]. However, it is not clear whether such mechanisms are scalable for DSM systems in which memory access latencies are often large and unpredictable.

1.1.3 Architecture-Oriented Memory Models

Many relaxed memory models have been proposed for DSM systems and software DSM systems. Weak consistency [37, 106] assumes that memory accesses to shared variables are guarded by synchronizations, and allows memory accesses between synchronizations to be performed out-of-order and interleaved with each other. A synchronization operation cannot be performed until all preceding ordinary memory accesses are completed; an ordinary memory access cannot be performed until all preceding synchronization operations are completed.

Release consistency [50, 49] further classifies synchronizations as acquire and release operations. Before an ordinary load or store access is allowed to perform with respect to any other processors, all previous acquire accesses must be performed. Before a release access is allowed to perform with respect to another processor, all previous ordinary load and store accesses must be performed. Release consistency allows non-atomic memory accesses since the execution of memory accesses between acquire and release operations does not have to be visible immediately to other processors. Release consistency was first implemented in the DASH system [79, 78].

The essence of release consistency is that memory accesses before a release must be globally performed before the synchronization lock can be released. Lazy release consistency [68] goes a step further. It allows a synchronization lock to be released to another processor even before previous memory accesses have been globally performed, provided the semantic effect of those memory accesses has become observable to the processor about to acquire the lock. This allows coherence actions to be postponed to the acquire point, while the release operation may involve no coherence action so that unnecessary coherence communication can be avoided. Again it can be shown that properly synchronized programs execute correctly under both release consistency and lazy release consistency, allowing more flexibility in implementations.

Entry consistency [16] can also be viewed as an extension of release consistency on another dimension. While each semaphore guards all shared variables in release consistency, entry consistency requires an explicit correspondence between each semaphore and the shared variables it guards. Consequently, only the shared data guarded by the corresponding lock need to be consistent at an acquire point. This has profound impact on programming since the the correspondence between locks and guarded shared variables must be explicit to the compiler and run-time system. Memory models such as scope consistency [60] have also been proposed to combine release consistency and entry consistency.

Location consistency [44, 45] models the state of a memory location as a partially ordered multiset of write and synchronization operations. Two write operations are ordered if they are performed on the same processor; a write operation and a synchronization operation are ordered if the processor that performs the write operation also participates in the synchronization. For each read operation, there is a set of legal values that can be supplied. Since processors do not have to observe the same ordering of write operations on the same memory location, a cache coherence protocol that obeys location consistency can allow a write operation to be performed without the exclusive ownership of the memory block.

1.1.4 Program-Oriented Memory Models

Even though sequential consistency is easy to understand, it needs to be augmented with semaphores to make parallel programming easier. Semaphores are atomic operations to acquire and release locks, which guard every shared writable variable. One of the powers of sequential consistency is that the lock operations can be defined in the model itself. When used properly, locks ensure that only one processor at a time can access a shared variable (though it is often safe to read a shared variable without acquiring the lock).

It is common practice in parallel programming to use semaphores to synchronize concurrent processes or threads properly to avoid data races. A data race occurs when there are multiple concurrent accesses to a shared variable, at least one of which is a write operation. Intuitively, a program is properly-synchronized if it contains enough synchronization operations so that conflicting memory accesses are always ordered by synchronization operations (that is, data races are limited to acquiring semaphores).

It is possible to relax some constraints of sequential consistency in many sections of properly synchronized programs. Program-oriented memory models allow memory accesses to be reordered and store operations to be performed in some non-atomic fashion based on the common intuition that properly-synchronized programs can behave sequentially consistent on an architecture supporting some relaxed memory model. Although it is generally undecidable if a program is properly-synchronized, it is often relatively convenient for the programmer to characterize each memory operation as ordinary or synchronization access.

Program-oriented memory models can be often defined in terms of some synchronization constraints on the program. The system guarantees to appear sequentially consistent provided that the program conforms to the synchronization constraints [2, 3, 47]. The intuition behind is that we can ensure sequential consistency for properly-synchronized programs by just placing coherence restrictions on synchronizations. Therefore, the programmer can rely on sequential consistency to reason about the program, although the memory access operations are performed in some out-of-order and non-atomic fashion.

Program-oriented memory models usually require that ordinary memory access operations be distinguished from synchronization operations. Furthermore, different synchronization operations can be exposed in different forms, which can have different coherence implications on implementations. Synchronization operations can be classified as acquire and release operations, loop and non-loop operations, and so on. Based on such classification, notions such as Data-Race-Free programs [2, 3] and Properly-Labeled programs [47, 46] have been developed.

Data-Race-Free-0 [2] only draws distinction between ordinary accesses and synchronizations. A program obeys Data-Race-Free-0 if, for any sequentially consistent execution, all conflicting memory accesses are ordered by the transitive relation of program order and synchronization order. Data-Race-Free-1 [3] furthermore characterizes paired synchronizations as acquire and release operations. The Properly-Labeled programs [46, 47] can classify competing accesses as loop and non-loop accesses, requiring the identification of a common class of synchronization accesses where one processor repeatedly reads a location until another processor writes a particular value to that location.

Dag consistency [17, 18] is a memory model that is defined on the directed acyclic graph (dag) of user-level threads that make up a parallel computation. Intuitively, each thread observes values that are consistent with some serial execution order of the dag, but different threads can observe different serial orders. Thus, store operations performed by a thread can be observed by its successors, but threads that are incomparable in the dag may or may not observe each other's write operations. Dag consistency allows load operations to retrieve values that are based on different sequential executions, provided that data dependencies of the dag are respected.

In this thesis, we propose the Commit-Reconcile & Fences (CRF) model, a mechanism-oriented memory model that is intended for architects and compiler writers. The CRF model has a semantic notion of caches which makes the operational behavior of data replication to be part of the model. It decomposes memory access operations into some finer-grain instructions: a load operation becomes a Reconcile followed by a Loadl, and a store operation becomes a Storel followed by a Commit. A novel feature of CRF is that programs written under various memory models such as sequential consistency and release consistency can be translated into efficient CRF programs. With appropriate extension, CRF can also be used to define and improve program-oriented memory models such as the Java memory model [85].

1.2 Cache Coherence Protocols

Shared memory multiprocessor systems provide a global address space in which processors can exchange information and synchronize with one another. When shared variables are cached in multiple caches simultaneously, a memory store operation performed by one processor can make data copies of the same variable in other caches out of date. The primary objective of a cache coherence protocol is to provide a coherent memory image for the system so that each processor can observe the semantic effect of memory access operations performed by other processors in time.

Shared memory systems can be implemented using different approaches. Typical hardware implementations extend traditional caching techniques and use custom communication interfaces and specific hardware support [5, 10, 20, 24, 79, 91, 93]. In Non-Uniform Memory Access (NUMA) systems, each site contains a memory-cache controller that determines if an access is to the local memory or some remote memory, based on the physical address of the memory access. In Cache Only Memory Architecture (COMA) systems, each local memory behaves as a large cache that allows shared data to be replicated and migrated. Simple COMA (S-COMA) systems divide the task of managing the global virtual address space between hardware and software to provide automatic data migration and replication with much less expensive hardware support. Hybrid systems can combine the advantages of NUMA and COMA systems [43, 65].

Software shared memory systems can employ virtual memory management mechanisms to achieve sharing and coherence [9, 81]. They often exhibit COMA-like properties by migrating and replicating pages between different sites. One attractive way to build scalable shared memory systems is to use small-scale to medium-scale shared memory machines as clusters that are interconnected with an off-the-shelf network [41, 126]. Such systems can effectively couple hardware cache coherence with software DSM systems. Shared memory systems can also be implemented via compilers that convert shared memory accesses into synchronization and coherence primitives [105, 107].

1.2.1 Snoopy Protocols and Directory-based Protocols

There are two types of cache coherence protocols: snoopy protocols for bus-based systems and directory-based protocols for DSM systems. In bus-based multiprocessor systems, since an ongoing bus transaction can be observed by all the processors, appropriate coherence actions can be taken when an operation threatening coherence is detected. Protocols that fall into this category are called snoopy protocols because each cache snoops bus transactions to watch memory transactions of other processors. Various snoopy protocols have been proposed [13, 38, 52, 53, 67, 96]. When a processor reads an address not in its cache, it broadcasts a read request on the snoopy bus. Memory or the cache that has the most up-to-date copy will then supply the data. When a processor broadcasts its intention to write an address which it does not own exclusively, other caches need to invalidate or update their copies.

Unlike snoopy protocols, directory-based protocols do not rely upon the broadcast mechanism to invalidate or update stale copies. They maintain a directory entry for each memory block to record the cache sites in which the memory block is currently cached. The directory entry is often maintained at the site in which the corresponding physical memory resides. Since the locations of shared copies are known, the protocol engine at each site can maintain coherence by employing point-to-point protocol messages. The elimination of broadcast overcomes a major limitation on scaling cache coherent machines to large-scale multiprocessor systems.

A directory-based cache coherence protocol can be implemented with various directory structures [6, 25, 26]. The full-map directory structure [77] maintains a complete record of which caches are sharing the memory block. In a straightforward implementation, each directory entry contain one bit per cache site representing if that cache has a shared copy. Its main drawback is that the directory space can be intolerable for large-scale systems. Alternative directory structures have been proposed to overcome this problem [27, 115, 118]. Different directory structures represent different implementation tradeoffs between performance and implementation complexity and cost [89, 94, 114].

A cache coherence protocol always implements some memory model that defines the semantics for memory access operations. Most snoopy protocols ensure sequential consistency, provided that memory accesses are performed in order. More sophisticated cache coherence protocols can implement relaxed memory models to improve performance.

1.2.2 Adaptive Cache Coherence Protocols

Shared memory programs have various access patterns [116, 124]. Empirical evidence suggests that no fixed cache coherence protocol works well for all access patterns [15, 38, 39, 42, 122]. For example, an invalidation-based MESI-like protocol assumes no correlation between processors that access the same address before and after a write operation; the protocol behaves as if the processor that modifies an address is likely to modify the same address again in near future. Needless to say, such a protocol is inappropriate for many common access patterns.

In shared memory systems, memory references can suffer long latencies for cache misses. To ameliorate this latency, a cache coherence protocol can be augmented with optimizations for different access patterns. Generally speaking, memory accesses can be classified into a number of common sharing patterns, such as the read-modify-write pattern, the producer-consumer pattern and the migratory pattern. An adaptive system can change its actions to address changing program behaviors. For example, Cox and Fowler [34] described an adaptive protocol that can dynamically identify migratory shared data in order to reduce the cost of moving them. Stenström et al. [119] proposed an adaptive protocol that can merge an invalidation request with a preceding cache miss request for migratory sharing. Lebeck and Wood [76] introduced dynamic self-invalidation to eliminate invalidation messages by having a processor automatically invalidate its local copy of a cache block before a conflicting access by another processor. Amza et al. [8] presented a software DSM system that can adapt between single-writer and multi-writer protocols, and allow dynamic data aggregation into larger transfer units.

Shared memory systems can provide flexible mechanisms that support various cache coherence protocols [10, 71]. A simple adaptive cache coherence protocol can incorporate multiple protocols in order to adapt to various identifiable access patterns. One major difference in these systems is regarding what and how access patterns are detected. Some heuristic mechanisms have been proposed to predict and trigger appropriate protocol actions [90]. Previous research shows that application-specific protocols can lead to performance improvement by tailoring cache coherence protocols to match specific communication patterns and memory semantics of applications [28, 42].

1.2.3 Verification of Cache Coherence Protocols

Verification of cache coherence protocols has gained considerable attention in recent years [4, 13, 21, 98, 99, 101, 102, 103, 120]. Most verification methods are based on state enumeration [63, 64] and symbolic model checking [30, 87], which can check correctness of assertions by exhaustively exploring all reachable states of the system. For example, Stern and Dill [120] used the Murφ system to automatically check if all reachable states satisfy certain properties which are attached to protocol specifications. Pong and Dubois [101] exploited the symmetry and homogeneity of the system states by keeping track of whether zero, one or multiple copies have been cached. This can reduce the state space and make the verification independent of the number of processors. Generally speaking, the major difference among these techniques is the representation of protocol states and the pruning method adopted in the state expansion process. Exponential state explosion has been a serious concern for model checker approaches, although various techniques have been proposed to reduce the state space.

While finite-state verifiers can be used for initial sanity checking on small scale examples, theorem provers can be of great help for verification of sophisticated protocols. Akhiani et al. [7] employed a hierarchical proof technique to verify sophisticated cache coherence protocols for the Alpha memory model. The protocols are specified in TLA+ [74, 75], a formal specification language based on first-order logic and set theory. Plakal et al. [31, 100] proposed a technique based on Lamport's logical clocks that can be used to reason about cache coherence protocols. The method associates a counter with each host and provides a time-stamping scheme that totally orders all protocol events. The total order can then be used to verify that the requirements of specific memory models are satisfied.

Most protocol verification methods verify certain invariants for cache coherence protocols. However, it is often difficult to determine all the necessary invariants in a systematic manner, especially for sophisticated protocols that implement relaxed memory models and incorporate various optimizations. While some invariants are obvious (for example, two caches at the same level should not contain the same address in the exclusive state simultaneously), many others are motivated by particular protocol implementations instead of the specifications of memory models. Sometimes it is not even clear if the chosen invariants are necessary or sufficient for the correctness. This means that for the same memory model, we may have to prove very different properties for different implementations. Therefore, these techniques are more like a bag of useful tools for debugging cache coherence protocols, rather than for verifying them.

The difficulty of protocol verification with current approaches can be largely attributed to the fact that protocols are designed and verified separately. In our approach, both the memory model and the protocol are expressed in the same formalism, and there is a notion that one system implements another. We begin with the operational specification of the memory model, and then develop protocols using the Imperative-&-Directive design methodology. Protocols are designed and verified iteratively throughout the successive process. The invariants that need to be verified usually show up systematically as lemmas that can be verified by induction and case analysis on rewriting rules.

1.3 Contributions of the Thesis

This thesis presents a mechanism-oriented memory model and associated adaptive cache coherence protocols that implement the memory model for DSM systems. The major contributions of the thesis are as follows:

- Using TRSs to model computer architectures and distributed protocols. TRSs offer a convenient way to describe asynchronous systems and can be used to prove the correctness of an implementation with respect to a specification. To prove its soundness, the specification is shown to be able to simulate the implementation with respect to a mapping function based on the notion of drained terms. To prove its liveness, temporal logic is employed to reason about time-varying behaviors of TRSs.

- The CRF memory model that allows great implementation flexibility for scalable DSM systems. The CRF model exposes a semantic notion of caches, referred to as saches, and decomposes memory load and store operations into finer-grain instructions. Programs written under sequential consistency and various relaxed memory models can be translated into efficient CRF programs. The precise definition of CRF can be used to verify cache coherence protocols.
- The two-stage Imperative-&-Directive methodology that can dramatically simplify protocol design and verification by separating the soundness and liveness concerns throughout protocol development. The first stage involves imperative rules that determine the soundness, and the second stage integrates directive rules with imperative rules to ensure the liveness. The integrated rules can be classified into mandatory rules and voluntary rules. Voluntary rules can provide adaptive actions without specific triggering conditions.
- Cache coherence protocols that implement the CRF model for DSM systems. The protocols are optimized for different access patterns, and are distinctive in the actions performed while committing dirty cache cells and reconciling clean cache cells. We prove that each protocol is a correct implementation of CRF in that it only exhibits behaviors permitted by the memory model, and is free from any type of deadlock and livelock.
- A cache coherence protocol called Cachet that provides enormous adaptivity for programs with different access patterns. The Cachet protocol is a seamless integration of multiple protocols, and incorporates both intra-protocol and inter-protocol adaptivity that can be exploited via appropriate heuristic mechanisms to achieve optimal performance under changing program behaviors.

The remainder of this thesis is logically composed of four parts. The first part, Chapter 2, introduces TRS and its application in verification of computer systems. As an example, we show that a processor with register renaming and speculative execution is a correct implementation of a simple instruction set specified by an in-order processor. The speculative processor also provides examples of memory instruction dispatch rules for the memory models and cache coherence protocols that are presented in this thesis.

In the second part, Chapter 3, we define the CRF memory model by giving operational semantics of memory access instructions Loadl and Storel, memory rendezvous instructions Commit and Reconcile, and memory fence instructions. Some derived rules of CRF are discussed, and we show that many memory models can be described as restricted versions of CRF. We also present a generalized version of CRF that allows the semantic effect of a memory store operation to be observed by different processors in some non-atomic fashion.

The third part of the thesis, Chapters 4, 5, and 6, describes several cache coherence protocols that implement CRF for DSM systems. In Chapter 4, we propose the Imperative-&-Directive design methodology, and present the Base protocol, a straightforward implementation of CRF. We prove its soundness by showing that each Base rule can be simulated in CRF, and its liveness by showing that each processor can always make progress. We develop directory-based cache coherence protocols WP and Migratory in Chapters 5 and 6, respectively.

In the fourth part of the thesis, Chapter 7, an adaptive cache coherence protocol called Cachet is designed through the integration of the Base, WP and Migratory protocols. Downgrade and upgrade operations are further discussed that allow a cache to switch from one protocol to another dynamically. We also demonstrate that the categorization of protocol messages into basic and composite messages can remarkably simplify the design and verification of cache coherence protocols. Finally, in Chapter 8, a conclusion is drawn and some future work is proposed.

Chapter 2

Using TRS to Model Architectures

Term Rewriting Systems (TRSs) offer a convenient way to describe parallel and asynchronous systems, and can be used to prove an implementation's correctness with respect to a specification. The state of a system is represented as a TRS term while the state transitions are represented as TRS rules. Although TRSs have been used extensively in programming language research to give operational semantics, their use in architectural descriptions is novel. TRSs can describe both deterministic and non-deterministic computations.

In microprocessors and memory systems, several actions may occur asynchronously. These systems are not amenable to sequential descriptions because sequentiality either causes over-specification or does not allow consideration of situations that may arise in real implementations. TRSs provide a natural way to describe such systems. Other formal techniques, such as Lamport's TLA [73] and Lynch's I/O automata [84], also enable us to model asynchronous systems. While all these techniques have something in common with TRSs, we find the use of TRSs more intuitive in both architecture descriptions and correctness proofs.

This chapter presents a brief introduction about TRSs.[1] We show that TRSs can be used to prove the correctness of an implementation with respect to a specification. As an example, we use TRSs to describe a speculative processor capable of register renaming and speculative execution, and demonstrate that the speculative processor produces the same set of behaviors as a simple non-pipelined implementation. The clarity of TRS descriptions allows us to model optimizing features such as write buffers and non-blocking caches.

2.1 Term Rewriting Systems

A term rewriting system is defined as a tuple (S, R, S_0), where S is a set of terms, R is a set of rewriting rules, and S_0 is a set of initial terms ($S_0 \subseteq S$). The state of a system is represented as a TRS term, while the state transitions are represented as TRS terms. The general structure of rewriting rules is as follows:

[1] Part of the chapter was first published in IEEE Micro Special Issue on "Modeling and Validation of Microprocessors", May/June 1999. We acknowledge IEEE for the reprint permission.

$$\begin{array}{ll} & s_1 \quad \textit{if} \quad p\,(s_1) \\ \rightarrow & s_2 \end{array}$$

where s_1 and s_2 are terms, and p is a predicate.

We can use a rule to rewrite a term if the rule's left-hand-side pattern matches the term or one of its subterms and the corresponding predicate is true. The new term is generated in accordance with the right-hand-side of the rule. If several rules apply, then any one of them can be applied. If no rule applies, then the term cannot be rewritten any further and is said to be in *normal form*. In practice, we often use abstract data types such as arrays and FIFO queues to make the descriptions more readable.

As an example, the Euclides algorithm for computing the greatest common divisor of two numbers can be expressed as follows:

$$\begin{array}{llll} \mathrm{GCD}(x,y) & \textit{if} \;\; x < y & \rightarrow & \mathrm{GCD}(y,x) \\ \mathrm{GCD}(x,y) & \textit{if} \;\; x \geq y \wedge y \neq 0 & \rightarrow & \mathrm{GCD}(x\text{-}y,y) \end{array}$$

The SK combinatory system, which has only two rules and a simple grammar for generating terms, provides a small but fascinating example of term rewriting. These two rules are sufficient to describe any computable function.

$$\begin{array}{llll} \textit{K-rule:} & (\mathrm{K}.x).y & \rightarrow & x \\ \textit{S-rule:} & ((\mathrm{S}.x).y).z & \rightarrow & (x.z).(y.z) \end{array}$$

We can verify that, for any subterm x, the term ((S.K).K).x can be rewritten to (K.x).(K.x) by applying the S-rule. We can rewrite the term further to x by applying the K-rule. Thus, if we read the dot as a function application, then the term ((S.K).K) behaves as the identity function. Notice the S-rule rearranges the dot and duplicates the term represented by x on the right-hand-side. For architectures in which terms represent states, rules must be restricted so that terms are not restructured or duplicated as in the S and K rules.

We say term s_1 can be rewritten to term s_2 in zero or more steps ($s_1 \twoheadrightarrow s_2$) if s_1 and s_2 are identical or there exists a term s' such that s_1 can be rewritten to s' in one step ($s_1 \rightarrow s'$) and s' can be rewritten to s_2 in zero or more steps ($s' \twoheadrightarrow s_2$). A TRS is confluent if for any term s_1, if $s_1 \twoheadrightarrow s_2$ and $s_1 \twoheadrightarrow s_3$, then there exists a term s_4 such that $s_2 \twoheadrightarrow s_4$ and $s_3 \twoheadrightarrow s_4$. A TRS is strongly terminating if a term can always be rewritten to a normal form regardless of the order in which the rules are applied. More information about TRSs can be found elsewhere [14, 69].

2.2 The AX Instruction Set

We use AX, a minimalist RISC instruction set, to illustrate all the processor examples in this chapter. The TRS description of a simple AX architecture also provides a good introductory example to the TRS notation. In the AX instruction set (Figure 2.1), all arithmetic operations

INST	$\equiv$	r:=Loadc(v)	*Load-constant Instruction*
	[]	r:=Loadpc	*Load-program-counter Instruction*
	[]	r:=Op(r_1,r_2)	*Arithmetic-operation Instruction*
	[]	Jz(r_1,r_2)	*Branch Instruction*
	[]	r:=Load(r_1)	*Load Instruction*
	[]	Store(r_1,r_2)	*Store Instruction*

Figure 2.1: AX: A Minimalist RISC Instruction Set

are performed on registers, and only the Load and Store instructions are allowed to access memory.

We use 'r' to represent a register name, 'v' a value, 'a' a data memory address and 'ia' an instruction memory address. An identifier may be qualified with a subscript. We do not specify the number of registers, the number of bits in a register or value, or the exact bit-format of each instruction. Such details are not necessary for a high-level description of a micro-architecture but are needed for synthesis. To avoid unnecessary complications, we assume that the instruction address space is disjoint from the data address space, so that self-modifying code is forbidden.

Semantically, AX instructions are executed strictly according to the program order: the program counter is incremented by one each time an instruction is executed except for the Jz instruction, where the program counter is set appropriately according to the branch condition. The informal meaning of the instructions is as follows:

The load-constant instruction r:=Loadc(v) puts constant v into register r. The load-program-counter instruction r:=Loadpc puts the program counter's content into register r. The arithmetic-operation instruction r:=Op(r_1,r_2) performs the arithmetic operation specified by Op on the operands specified by registers r_1 and r_2 and puts the result into register r. The branch instruction Jz(r_1,r_2) sets the program counter to the target instruction address specified by register r_2 if register r_1 contains value zero; otherwise the program counter is simply increased by one. The load instruction r:=Load(r_1) reads the memory cell specified by register r_1, and puts the data into register r. The store instruction Store(r_1,r_2) writes the content of register r_2 into the memory cell specified by register r_1.

It is important to distinguish between variables and constants in pattern matching. A variable matches any expression while a constant matches only itself. We use the following notational conventions in the rewriting rules: all the special symbols such as ':=', and all the identifiers that start with capital letters are treated as constants in pattern matching. We use '-' to represent the wild-card term that can match any term. The grammar uses '[]' as a meta-notation to separate disjuncts.

We define the operational semantics of AX instructions using the P_{B} (Base Processor) model, a single-cycle, non-pipelined, in-order execution processor. Figure 2.2 shows the datapath for

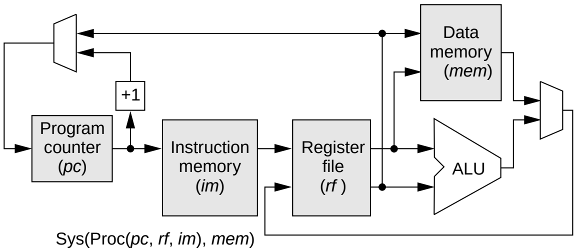


Figure 2.2: P_B: A Single-Cycle In-Order Processor

such a system. The processor consists of a program counter (*pc*), a register file (*rf*), and an instruction memory (*im*). The program counter holds the address of the instruction to be executed. The processor together with the data memory (*mem*) constitutes the whole system, which can be represented as the TRS term Sys(Proc(*pc*, *rf*, *im*), *mem*). The semantics of each instruction can be given as a rewriting rule which specifies how the state is modified after each instruction is executed.

Note that *pc*, *rf*, *im* and *mem* can be grouped syntactically in any convenient way. Grouping them as Sys(Proc(*pc*, *rf*, *im*), *mem*) instead of Sys(*mem*, *pc*, *rf*, *im*) provides a degree of modularity in describing the rules that do not refer to *mem*. Abstract data types can also enhance modularity. For example, *rf*, *im* and *mem* are all represented using arrays on which only two operations, selection and update, can be performed. Thus, $rf[r]$ refers to the content of register r, and $rf[r{:=}v]$ represents the register file after register r has been updated with value v. Similarly, $mem[a]$ refers to the content of memory location a, and $mem[a{:=}v]$ represents the memory with location a updated with value v. Notation $\underline{\text{Op}}(v_1,v_2)$ represents the result of operation Op with operands v_1 and v_2.

Loadc Rule

$$\begin{array}{lll} & \text{Proc}(ia,\ rf,\ im) & \textit{if}\quad im[ia] = r{:=}\text{Loadc}(v) \\ \rightarrow & \text{Proc}(ia{+}1,\ rf[r{:=}v],\ im) & \end{array}$$

Loadpc Rule

$$\begin{array}{lll} & \text{Proc}(ia,\ rf,\ im) & \textit{if}\quad im[ia] = r{:=}\text{Loadpc} \\ \rightarrow & \text{Proc}(ia{+}1,\ rf[r{:=}ia],\ im) & \end{array}$$

Op Rule

$$\begin{array}{lll} & \text{Proc}(ia,\ rf,\ im) & \textit{if}\quad im[ia] = r{:=}\text{Op}(r_1,r_2) \\ \rightarrow & \text{Proc}(ia{+}1,\ rf[r{:=}v],\ im) & \textit{where}\quad v = \underline{\text{Op}}(rf[r_1],rf[r_2]) \end{array}$$

Jz-Jump Rule

$$\begin{array}{lll} & \text{Proc}(ia,\ rf,\ im) & \textit{if}\quad im[ia] = \text{Jz}(r_1,r_2)\ \wedge\ rf[r_1] = 0 \\ \rightarrow & \text{Proc}(rf[r_2],\ rf,\ im) & \end{array}$$

Jz-NoJump Rule

$$\begin{array}{lll} & \text{Proc}(ia,\ rf,\ im) & \textit{if}\quad im[ia] = \text{Jz}(r_1,r_2)\ \wedge\ rf[r_1] \neq 0 \\ \rightarrow & \text{Proc}(ia{+}1,\ rf,\ im) & \end{array}$$

Current State: Sys(Proc(ia, rf, im), mem)				
Rule Name	**Instruction at ia**	**Next pc**	**Next rf**	**Next mem**
Loadc	r:=Loadc(v)	ia+1	$rf[r{:=}v]$	mem
Loadpc	r:=Loadpc	ia+1	$rf[r{:=}ia]$	mem
Op	r:=Op(r_1,r_2)	ia+1	$rf[r{:=}\underline{\text{Op}}(rf[r_1],rf[r_2])]$	mem
Jz	Jz(r_1,r_2)	ia+1 (if $rf[r_1] \neq 0$) $rf[r_2]$ (if $rf[r_1] = 0$)	rf	mem
Load	r:=Load(r_1)	ia+1	$rf[r{:=}mem[rf[r_1]]]$	mem
Store	Store(r_1,r_2)	ia+1	rf	$mem[rf[r_1]{:=}rf[r_2]]$

Figure 2.3: Operational Semantics of AX

Load Rule

$$\begin{array}{ll} & \text{Sys(Proc(}ia,\ rf,\ im\text{)},\ mem\text{)} \quad \textit{if} \quad im[ia] = \ r{:=}\text{Load}(r_1) \\ \to & \text{Sys(Proc(}ia{+}1,\ rf[r{:=}mem[a]],\ im\text{)},\ mem\text{)} \qquad \textit{where} \quad a = rf[r_1] \end{array}$$

Store Rule

$$\begin{array}{ll} & \text{Sys(Proc(}ia,\ rf,\ im\text{)},\ mem\text{)} \quad \textit{if} \quad im[ia] = \ \text{Store}(r_1,r_2) \\ \to & \text{Sys(Proc(}ia{+}1,\ rf,\ im\text{)},\ mem[a{:=}rf[r_2]]\text{)} \qquad \textit{where} \quad a = rf[r_1] \end{array}$$

Since the pattern Proc(ia, rf, im) will match any processor term, the real discriminant is the instruction at address ia. In the case of a branch instruction, further discrimination is made based on the value of the condition register. Figure 2.3 summarizes the P_B rules given above. Given proper context, it should be easy to deduce the precise TRS rules from a tabular description.

It is important to understand the atomic nature of these rules. Once a rule is applied, the state specified by its right-hand-side must be reached before any other rule can be applied. For example, on an Op instruction, both operands must be fetched and the result computed and stored in the register file in one atomic action. Furthermore, the program counter must be updated during this atomic action as well. This is why these rules describe a single-cycle, non-pipelined implementation of AX.

2.3 Register Renaming and Speculative Execution

Many possible micro-architectures can implement the AX instruction set. For example, in a simple pipelined architecture, instructions are fetched, executed and retired in order, and the processor can contain as many as four or five partially executed instructions. Storage in the form of pipeline buffers is provided to hold these partially executed instructions. More sophisticated pipelined architectures have multiple functional units that can be specialized for integer or floating-point calculations. In such architectures, instructions issued in order may nevertheless complete out of order because of varying functional unit latencies. An implementation preserves correctness by ensuring that a new instruction is not issued when there is another instruction in the pipeline that may update any register to be read or written by the new instruction. Cray's CDC 6600, one of the earliest examples of such an architecture, used a scoreboard to dispatch

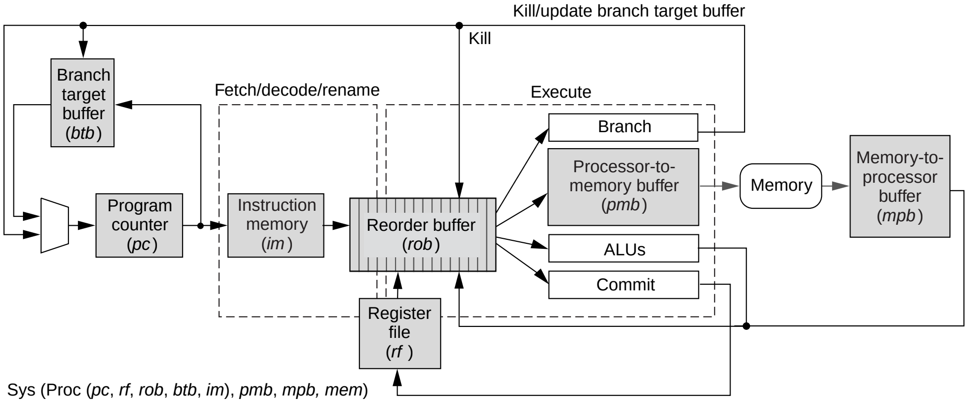

Kill/update branch target buffer
Kill
Branch target buffer (*btb*)
Fetch/decode/rename
Execute
Branch
Processor-to-memory buffer (*pmb*)
Memory
Memory-to-processor buffer (*mpb*)
Program counter (*pc*)
Instruction memory (*im*)
Reorder buffer (*rob*)
ALUs
Commit
Register file (*rf*)
Sys (Proc (*pc*, *rf*, *rob*, *btb*, *im*), *pmb*, *mpb, mem*)

Figure 2.4: $\mathrm{P_S}$: A Processor with Register Renaming and Speculative Execution

and track partially executed instructions in the processor. In Cray-style scoreboard design, the number of registers in the instruction set limits the number of instructions in the pipeline.

In the mid-sixties, Robert Tomasulo at IBM invented the technique of register renaming to overcome this limitation on pipelining. He assigned a renaming tag to each instruction as it was decoded. The following instructions used this tag to refer to the value produced by this instruction. A renaming tag became free and could be used again once the instruction was completed. The micro-architecture maintained the association between the register name, the tag and the associated value (whenever the value became available). This innovative idea was embodied in IBM 360/91 in the late sixties. By the mid-nineties, register renaming had become commonplace and is now present in all high-end microprocessors.

An important state element in a micro-architecture with register renaming is a reorder buffer (ROB), which holds instructions that have been decoded but have not completed their execution (see Figure 2.4). Conceptually, a ROB divides the processor into two asynchronous parts. The first part fetches an instruction and, after decoding and renaming registers, dumps it into the next available slot in the ROB. The ROB slot index serves as a renaming tag, and the instructions in the ROB always contain tags or values instead of register names. An instruction in the ROB can be executed if all its operands are available. The second part of the processor takes any enabled instruction out of the ROB and dispatches it to an appropriate functional unit, including the memory system. This mechanism is very similar to the execution mechanism in dataflow architectures. Such an architecture may execute instructions out of order, especially if functional units have different latencies or there are data dependencies between instructions.

In addition to register renaming, most contemporary microprocessors also permit speculative execution of instructions. On the base of the program's past behavior, the speculative mechanisms predict the address of the next instruction to be issued. The speculative instruction's address is determined by consulting a table known as the branch target buffer (BTB). The BTB can be indexed by the program counter. If the prediction turns out to be wrong, the speculative instruction and all the instructions issued thereafter are abandoned, and their effect on the processor state nullified. The BTB is updated according to some prediction scheme after each branch resolution.

The speculative processor's correctness is not contingent upon how the BTB is maintained, as long as the program counter can be set to the correct value after a misprediction. However, different prediction schemes can give rise to very different misprediction rates and thus profoundly influence performance. Generally, we assume that the BTB produces the correct next instruction address for all non-branch instructions. Any processor permitting speculative execution must ensure that a speculative instruction does not modify the programmer-visible state until it can be "committed". Alternatively, it must save enough of the processor state so that the correct state can be restored in case the speculation turns out to be wrong. Most implementations use a combination of these two ideas: speculative instructions do not modify the register file or memory until it can be determined that the prediction is correct, but they may update the program counter. Both the current and the speculated instruction address are recorded. Thus, speculation correctness can be determined later, and the correct program counter can be restored in case of a wrong prediction. Typically, all the temporary state is maintained in the ROB itself.

2.4 The $\mathrm{P_S}$ Speculative Processor

We now present the rules for a simplified micro-architecture that does register renaming and speculative execution. We achieve this simplification by not showing all the pipelining and not giving the details of some hardware operations. The memory system is modeled as operating asynchronously with respect to the processor. Thus, memory instructions in the ROB are dispatched to the memory system via an ordered processor-to-memory buffer (*pmb*); the memory provides its responses via a memory-to-processor buffer (*mpb*). Memory system details can be added in a modular fashion without changing the processor description.

We need to add two new components, *rob* and *btb*, to the processor state, corresponding to ROB and BTB. The reorder buffer is a complex device to model because different types of operations need to be performed on it. It can be thought of as a FIFO queue that is initially empty (ϵ). We use constructor '$\oplus$', which is associative but not commutative, to represent this aspect of *rob*. It can also be considered as an array of instruction templates, with an array index serving as a renaming tag. It is well known that a FIFO queue can be implemented as a circular buffer using two pointers into an array. We will hide these implementation details of *rob* and assume that the next available tag can be obtained.

An instruction template in *rob* contains the instruction address, opcode, operands and some extra information needed to complete the instruction. For instructions that need to update a register, the $\mathrm{Wr}(r)$ field records the destination register r. For branch instructions, the $\mathrm{Sp}(pia)$ field holds the predicted instruction address pia, which will be used to determine the prediction's correctness. Each memory access instruction maintains an extra flag to indicate whether the instruction is waiting to be dispatched (U), or has been dispatched to the memory (D). The memory system returns a value for a load and an acknowledgment (Ack) for a store. We have taken some syntactic liberties in expressing various types of instruction templates below:

$$
\begin{array}{lll}
\text{ROB Entry} & \equiv & \text{Itb}(ia,t{:=}v,\text{Wr}(r)) \\
 & [\!] & \text{Itb}(ia,t{:=}\text{Op}(tv_1,tv_2),\text{Wr}(r)) \\
 & [\!] & \text{Itb}(ia,\text{Jz}(tv_1,tv_2),\text{Sp}(pia)) \\
 & [\!] & \text{Itb}(ia,t{:=}\text{Load}(tv_1,mf),\text{Wr}(r)) \\
 & [\!] & \text{Itb}(ia,t{:=}\text{Store}(tv_1,tv_2,mf))
\end{array}
$$

where tv stands for either a tag or a value, and the memory flag mf is either U or D. The tag used in the store instruction template is intended to provide some flexibility in coordinating with the memory system and does not imply any register updating.

2.4.1 Instruction Fetch Rules

Each time the processor issues an instruction, the program counter is set to the address of the next instruction to be issued. For non-branch instructions, the program counter is simply incremented by one. Speculative execution occurs when a Jz instruction is issued: the program counter is then set to the instruction address obtained by consulting the *btb* entry corresponding to the Jz instruction's address.

When the processor issues an instruction, an instruction template is allocated in the *rob*. If the instruction is to modify a register, we use an unused renaming tag (typically the index of the *rob* slot) to rename the destination register, and the destination register is recorded in the Wr field. The tag or value of each operand register is found by searching the *rob* from the youngest (rightmost) buffer to the oldest (leftmost) buffer until an instruction template containing the referenced register is found. If no such buffer exists in the *rob*, then the most up-to-date value resides in the register file. The following lookup procedure captures this idea:

$$
\begin{array}{l}
\text{lookup}(r,\ rf,\ rob) \\
\qquad \equiv\ rf[r] \qquad \textit{if}\ \ \text{Wr}(r) \notin rob \\
\text{lookup}(r,\ rf,\ rob_1 \oplus \text{Itb}(ia,t{:=}\text{-},\text{Wr}(r)) \oplus rob_2) \\
\qquad \equiv\ t \qquad\quad \textit{if}\ \ \text{Wr}(r) \notin rob_2
\end{array}
$$

It is beyond the scope of our discussion to give a hardware implementation of this procedure, but it is certainly possible to do so using TRSs. Any implementation that can look up values in the *rob* using a combinational circuit will suffice.

Fetch-Loadc Rule

$$
\begin{array}{lll}
 & \text{Proc}(ia,\ rf,\ rob,\ btb,\ im) & \textit{if}\quad im[ia]\ =\ r{:=}\text{Loadc}(v) \\
\rightarrow & \text{Proc}(ia{+}1,\ rf,\ rob \oplus \text{Itb}(ia,t{:=}v,\text{Wr}(r)),\ btb,\ im) &
\end{array}
$$

Fetch-Loadpc Rule

$$
\begin{array}{lll}
 & \text{Proc}(ia,\ rf,\ rob,\ btb,\ im) & \textit{if}\quad im[ia]\ =\ r{:=}\text{Loadpc} \\
\rightarrow & \text{Proc}(ia{+}1,\ rf,\ rob \oplus \text{Itb}(ia,t{:=}ia,\text{Wr}(r)),\ btb,\ im) &
\end{array}
$$

Current State: Proc(ia, rf, rob, btb, im)			
Rule Name	**Instruction at ia**	**New Template in rob**	**Next pc**
Fetch-Loadc	r:=Loadc(v)	Itb(ia,t:=v,Wr(r))	ia+1
Fetch-Loadpc	r:=Loadpc	Itb(ia,t:=ia,Wr(r))	ia+1
Fetch-Op	r:=Op(r_1,r_2)	Itb(ia,t:=Op(tv_1,tv_2),Wr(r))	ia+1
Fetch-Jz	Jz(r_1,r_2)	Itb(ia,Jz(tv_1,tv_2),Sp($btb[ia]$))	$btb[ia]$
Fetch-Load	r:=Load(r_1)	Itb(ia,t:=Load(tv_1,U),Wr(r))	ia+1
Fetch-Store	Store(r_1,r_2)	Itb(ia,t:=Store(tv_1,tv_2,U))	ia+1

Figure 2.5: $\mathrm{P_S}$ Instruction Fetch Rules

Fetch-Op Rule

$$\begin{array}{ll} & \mathrm{Proc}(ia,\ rf,\ rob,\ btb,\ im) \quad \textit{if} \quad im[ia] = r{:=}\mathrm{Op}(r_1,r_2) \\ \rightarrow & \mathrm{Proc}(ia{+}1,\ rf,\ rob \oplus \mathrm{Itb}(ia,t{:=}\mathrm{Op}(tv_1,tv_2),\mathrm{Wr}(r)),\ btb,\ im) \end{array}$$

Fetch-Jz Rule

$$\begin{array}{ll} & \mathrm{Proc}(ia,\ rf,\ rob,\ btb,\ im) \quad \textit{if} \quad im[ia] = \mathrm{Jz}(r_1,r_2) \\ \rightarrow & \mathrm{Proc}(pia,\ rf,\ rob \oplus \mathrm{Itb}(ia,\mathrm{Jz}(tv_1,tv_2),\mathrm{Sp}(pia)),\ btb,\ im) \quad \textit{where} \quad pia = btb[ia] \end{array}$$

Fetch-Load Rule

$$\begin{array}{ll} & \mathrm{Proc}(ia,\ rf,\ rob,\ btb,\ im) \quad \textit{if} \quad im[ia] = r{:=}\mathrm{Load}(r_1) \\ \rightarrow & \mathrm{Proc}(ia{+}1,\ rf,\ rob \oplus \mathrm{Itb}(ia,t{:=}\mathrm{Load}(tv_1,\mathrm{U}),\mathrm{Wr}(r)),\ btb,\ im) \end{array}$$

Fetch-Store Rule

$$\begin{array}{ll} & \mathrm{Proc}(ia,\ rf,\ rob,\ btb,\ im) \quad \textit{if} \quad im[ia] = \mathrm{Store}(r_1,r_2) \\ \rightarrow & \mathrm{Proc}(ia{+}1,\ rf,\ rob \oplus \mathrm{Itb}(ia,t{:=}\mathrm{Store}(tv_1,tv_2,\mathrm{U})),\ btb,\ im) \end{array}$$

As an example of instruction fetch rules, consider the *Fetch-Op* rule, which fetches an Op instruction and after register renaming simply puts it at the end of the *rob*. In all the instruction fetch rules, t represents an unused tag, tv_1 and tv_2 represent the tag or value corresponding to the operand registers r_1 and r_2, respectively, that is, tv_1 = lookup(r_1, rf, rob), tv_2 = lookup(r_2, rf, rob). Figure 2.5 summarizes the instruction fetch rules.

Any implementation includes a finite number of *rob* entries, and the instruction fetch must be stalled if *rob* is full. This availability checking can be easily modeled. A fast implementation of the lookup procedure in hardware is quite difficult. Often a renaming table that retains the association between a register name and its current tag is maintained separately.

2.4.2 Arithmetic Operation and Value Propagation Rules

The arithmetic operation rule states that an arithmetic operation in the *rob* can be performed if both operands are available. It assigns the result to the corresponding tag. Note that the instruction can be in any position in the *rob*. The forward rule sends a tag's value to other instruction templates, while the commit rule writes the value produced by the oldest instruction in the *rob* to the destination register and retires the corresponding renaming tag. Notation $rob_2[v/t]$ means that one or more appearances of tag t in rob_2 are replaced by value v. Figure 2.6 summarizes the rules for arithmetic operation and value propagation.

Current State: Proc(ia, rf, rob, btb, im)			
Rule Name	**rob**	**Next rob**	**Next rf**
Op	$rob_1 \oplus$ Itb(ia_1,t:=Op(v_1,v_2),Wr(r))$\oplus$ rob_2	$rob_1 \oplus$ Itb(ia_1,t:= $\underline{\text{Op}}$(v_1,v_2),Wr(r))$\oplus$ rob_2	rf
Value-Forward	$rob_1 \oplus$ Itb(ia_1,t:=v,Wr(r))$\oplus$ rob_2 ($t \in rob_2$)	$rob_1 \oplus$ Itb(ia_1,t:=v,Wr(r))$\oplus$ $rob_2[v/t]$	rf
Value-Commit	Itb(ia_1,t:=v,Wr(r)) $\oplus$ rob ($t \notin rob$)	rob	$rf[r$:=$v]$

Figure 2.6: P_S Arithmetic Operation and Value Propagation Rules

Op Rule

  Proc(ia, rf, $rob_1 \oplus$ Itb(ia_1,t:=Op(v_1,v_2),Wr(r))$\oplus$ rob_2, btb, im)
$\rightarrow$ Proc(ia, rf, $rob_1 \oplus$ Itb(ia_1,t:=v,Wr(r))$\oplus$ rob_2, btb, im)   *where* $v = \underline{\text{Op}}(v_1, v_2)$

Value-Forward Rule

  Proc(ia, rf, $rob_1 \oplus$ Itb(ia_1,t:=v,Wr(r))$\oplus$ rob_2, btb, im)   *if* $t \in rob_2$
$\rightarrow$ Proc(ia, rf, $rob_1 \oplus$ Itb(ia_1,t:=v,Wr(r))$\oplus$ $rob_2[v/t]$, btb, im)

Value-Commit Rule

  Proc(ia, rf, Itb(ia_1,t:=v,Wr(r)) $\oplus$ rob, btb, im)   *if* $t \notin rob$
$\rightarrow$ Proc(ia, $rf[r$:=$v]$, rob, btb, im)

The *rob* pattern in the commit rule dictates that the register file can only be modified by the oldest instruction after it has forwarded the value to all the buffers in the *rob* that reference its tag. Restricting the register update to just the oldest instruction in the *rob* eliminates output (write-after-write) hazards, and protects the register file from being polluted by incorrect speculative instructions. It also provides a way to support precise interrupts. The commit rule is needed to free up resources and to let the following instructions reuse the tag.

2.4.3 Branch Completion Rules

The branch completion rules determine if the branch prediction was correct by comparing the predicted instruction address (pia) with the resolved branch target instruction address (nia or ia_1+1). If they do not match (meaning the speculation was wrong), all instructions issued after the branch instruction are aborted, and the program counter is set to the new branch target instruction. The branch target buffer btb is updated according to some prediction algorithm; btb' represents the new branch target buffer. Figure 2.7 summarizes the branch resolution cases. It is worth noting that the branch rules allow branches to be resolved in any order.

Jump-CorrectSpec Rule

  Proc(ia, rf, $rob_1 \oplus$ Itb(ia_1,Jz(0,nia),Sp(pia))$\oplus$ rob_2, btb, im)   *if* $pia = nia$
$\rightarrow$ Proc(ia, rf, $rob_1 \oplus rob_2$, btb', im)

Jump-WrongSpec Rule

  Proc(ia, rf, $rob_1 \oplus$ Itb(ia_1,Jz(0,nia),Sp(pia))$\oplus$ rob_2, btb, im)   *if* $pia \neq nia$
$\rightarrow$ Proc(nia, rf, rob_1, btb', im)

NoJump-CorrectSpec Rule

  Proc(ia, rf, $rob_1 \oplus$ Itb(ia_1,Jz(v,-),Sp(pia))$\oplus$ rob_2, btb, im)   *if* $v \neq 0 \ \wedge \ pia = ia_1 + 1$
$\rightarrow$ Proc(ia, rf, $rob_1 \oplus rob_2$, btb', im)

Current State: Proc(ia, rf, rob, btb, im)			
Rule Name	**rob** = $rob_1 \oplus \text{Itb}(ia_1,\text{Jz}(0,nia),\text{Sp}(pia)) \oplus rob_2$	**Next rob**	**Next pc**
Jump-CorrectSpec	$pia = nia$	$rob_1 \oplus rob_2$	ia
Jump-WrongSpec	$pia \neq nia$	rob_1	nia
Rule Name	**rob** = $rob_1 \oplus \text{Itb}(ia_1,\text{Jz}(v,\text{-}),\text{Sp}(pia)) \oplus rob_2$	**Next rob**	**Next pc**
NoJump-CorrectSpec	$v \neq 0,\ pia = ia_1+1$	$rob_1 \oplus rob_2$	ia
NoJump-WrongSpec	$v \neq 0,\ pia \neq ia_1+1$	rob_1	ia_1+1

Figure 2.7: P_S Branch Completion Rules

Current State: Sys(Proc(ia, rf, $rob_1 \oplus itb \oplus rob_2$, btb, im), pmb, mpb, mem)				
Rule Name	**itb**	**pmb**	**Next itb**	**Next pmb**
Dispatch-Load	$\text{Itb}(ia_1,t{:=}\text{Load}(a,\text{U}),\text{Wr}(r))$ U, Jz $\notin rob_1$	pmb	$\text{Itb}(ia_1,t{:=}\text{Load}(a,\text{D}),\text{Wr}(r))$	$pmb;\langle t,\text{Load}(a)\rangle$
Dispatch-Store	$\text{Itb}(ia_1,t{:=}\text{Store}(a,v,\text{U}))$ U, Jz $\notin rob_1$	pmb	$\text{Itb}(ia_1,t{:=}\text{Store}(a,v,\text{D}))$	$pmb;\langle t,\text{Store}(a,v)\rangle$
Rule Name	**itb**	**mpb**	**Next itb**	**Next mpb**
Retire-Load	$\text{Itb}(ia_1,t{:=}\text{Load}(a,\text{D}),\text{Wr}(r))$	$\langle t,v\rangle \mid mpb$	$\text{Itb}(ia_1,t{:=}v,\text{Wr}(r))$	mpb
Retire-Store	$\text{Itb}(ia_1,t{:=}\text{Store}(a,v,\text{D}))$	$\langle t,\text{Ack}\rangle \mid mpb$	ϵ (deleted)	mpb

Figure 2.8: P_S Memory Instruction Dispatch and Completion Rules

NoJump-WrongSpec Rule

$$\text{Proc}(ia,\ rf,\ rob_1 \oplus \text{Itb}(ia_1,\text{Jz}(v,\text{-}),\text{Sp}(pia)) \oplus rob_2,\ btb,\ im) \qquad \textit{if} \quad v \neq 0 \ \wedge \ pia \neq ia_1+1$$
$$\rightarrow \quad \text{Proc}(ia_1+1,\ rf,\ rob_1,\ btb',\ im)$$

The branch resolution mechanism becomes slightly complicated if certain instructions that need to be killed are waiting for responses from the memory system or some functional units. In such a situation, killing may have to be postponed until rob_2 does not contain an instruction waiting for a response. (This is not possible for the rules that we have presented).

2.4.4 Memory Access Rules

Memory requests are sent to the memory system strictly in order. A request is sent only when there is no unresolved branch instruction in front of it. The dispatch rules flip the U bit to D and enqueue the memory request into the *pmb*. The memory system can respond to the requests in any order and the response is used to update the appropriate entry in the *rob*. The semicolon is used to represent an ordered queue and the vertical bar an unordered queue (that is, the connective is both commutative and associative). Figure 2.8 summarizes the memory instruction dispatch and completion rules.

Dispatch-Load Rule

$$\text{Sys}(\text{Proc}(ia,\ rf,\ rob_1 \oplus \text{Itb}(ia_1,t{:=}\text{Load}(a,\text{U}),\text{Wr}(r)) \oplus rob_2,\ btb,\ im),\ pmb,\ mpb,\ mem)$$
$$\textit{if} \quad \text{Jz, U} \notin rob_1$$
$$\rightarrow \quad \text{Sys}(\text{Proc}(ia,\ rf,\ rob_1 \oplus \text{Itb}(ia_1,t{:=}\text{Load}(a,\text{D}),\text{Wr}(r)) \oplus rob_2,\ btb,\ im),\ pmb;\langle t,\text{Load}(a)\rangle,\ mpb,\ mem)$$

Rule Name	**mem**	**pmb**	**mpb**	**Next mem**	**Next pmb**	**Next mpb**
Load-Memory	*mem*	$\langle t$,Load$(a)\rangle;pmb$	*mpb*	*mem*	*pmb*	$mpb|\langle t,mem[a]\rangle$
Store-Memory	*mem*	$\langle t$,Store$(a,v)\rangle;pmb$	*mpb*	$mem[a{:=}v]$	*pmb*	$mpb|\langle t$,Ack$\rangle$

Figure 2.9: A Simple Processor-Memory Interface

Dispatch-Store Rule

$\quad$ Sys(Proc(ia, rf, $rob_1\oplus$ Itb(ia_1,t:=Store(a,v,U))$\oplus$ rob_2, btb, im), pmb, mpb, mem)

$\qquad\qquad$ *if* $\quad$ Jz, U $\notin rob_1$

$\rightarrow$ Sys(Proc(ia, rf, $rob_1\oplus$ Itb(ia_1,t:=Store(a,v,D))$\oplus$ rob_2, btb, im), pmb;$\langle t$,Store(a,v)$\rangle$, mpb, mem)

Retire-Load Rule

$\quad$ Sys(Proc(ia, rf, $rob_1\oplus$ Itb(ia_1,t:=Load(a,D),Wr(r))$\oplus$ rob_2, btb, im), pmb, $\langle t,v\rangle|mpb$, mem)

$\rightarrow$ Sys(Proc(ia, rf, $rob_1\oplus$ Itb(ia_1,t:=v,Wr(r))$\oplus$ rob_2, btb, im), pmb, mpb, mem)

Retire-Store Rule

$\quad$ Sys(Proc(ia, rf, $rob_1\oplus$ Itb(ia_1,t:=Store(a,v,D))$\oplus$ rob_2, btb, im), pmb, $\langle t$,Ack$\rangle|mpb$, mem)

$\rightarrow$ Sys(Proc(ia, rf, $rob_1\oplus rob_2$, btb, im), pmb, mpb, mem)

We define a simple interface between the processor and the memory that ensures memory accesses are processed in order to guarantee sequential consistency in multiprocessor systems. Figure 2.9 summarizes the rules for how the memory system handles memory requests from the *pmb*. More aggressive implementations of memory access operations are possible, but they often lead to various relaxed memory models in multiprocessor systems.

Load-Memory Rule

$\quad$ Sys($proc$, $\langle t$,Load(a)$\rangle$;pmb, mpb, mem)

$\rightarrow$ Sys($proc$, pmb, $mpb|\langle t,mem[a]\rangle$, mem)

Store-Memory Rule

$\quad$ Sys($proc$, $\langle t$,Store(a,v)$\rangle$;pmb, mpb, mem)

$\rightarrow$ Sys($proc$, pmb, $mpb|\langle t$,Ack$\rangle$, $mem[a{:=}v]$)

2.5 Using TRS to Prove Correctness

TRSs offer a convenient way to describe asynchronous systems and can be used to prove the correctness of an implementation with respect to a specification. The correctness of an implementation can be described in terms of soundness and liveness properties. A soundness property ensures that the implementation cannot take an action that is inconsistent with the specification, that is, something bad does not happen. A liveness property ensures that the implementation makes progress eventually, that is, something good does eventually happen.

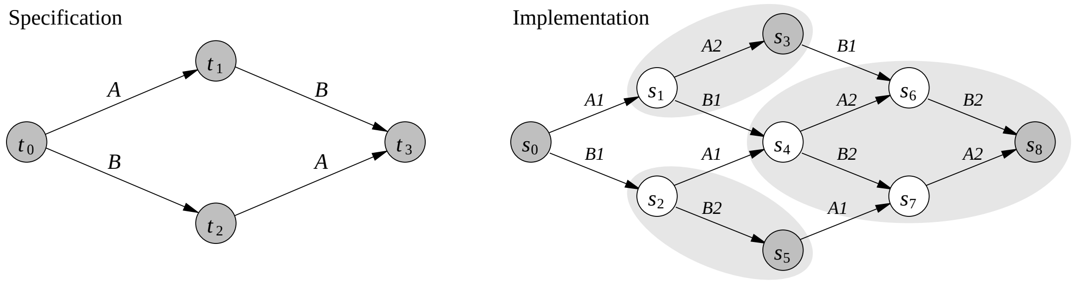


Figure 2.10: Forward Draining

2.5.1 Verification of Soundness

We use simulation to prove the soundness of an implementation. We first build a mapping function that maps an implementation term to a specification term, and then show that the specification can simulate the implementation with respect to the mapping function. We define the mapping function based on the notion of drained terms or drained states. Intuitively, a drained implementation term contains no partially executed operation and therefore can be trivially mapped to a specification term via a projection function (or a combination of a projection function and a lift function).

Based on the concept of drained terms, the implementation terms are classified into equivalent classes where each equivalent class contains exactly one drained term as the representative of the class. A term can be rewritten to a drained term via forward or backward draining. Forward draining means that the term can be rewritten to the corresponding drained term according to some implementation rules; backward draining means that the drained term can be rewritten back to the term itself according to some implementation rules. Intuitively, forward draining completes partially executed operations, while backward draining cancels partially executed operations and recovers the system state. We can specify a set of draining rules so that for each implementation term, its normal form with respect to the draining rules represents a drained term.

Figure 2.10 shows the use of forward draining. The specification allows t_0 to be rewritten to t_3 by applying rules A and B; the order in which the rules are applied does not matter. The implementation takes two consecutive steps to achieve the semantic effect of each specification rule: $A1$ and $A2$ for rule A, and $B1$ and $B2$ for rule B. The application of rules $A1$ and $A2$ can be interleaved with the application of rules $B1$ and $B2$. It is obvious that s_0, s_3, s_5 and s_8 are drained terms that correspond to t_0, t_1, t_2 and t_3, respectively. We can use forward draining to drain other implementation terms by completing partially executed operations; this can be achieved by chosing $A2$ and $B2$ as the draining rules. Therefore, s_1 is drained to s_3, s_2 is drained to s_5, and s_4, s_6 and s_7 are drained to s_8.

Figure 2.11 shows the use of backward draining in a non-confluent system. The specification allows t_0 to be rewritten to t_3 or t_4, while applying rules A and B in different orders can lead

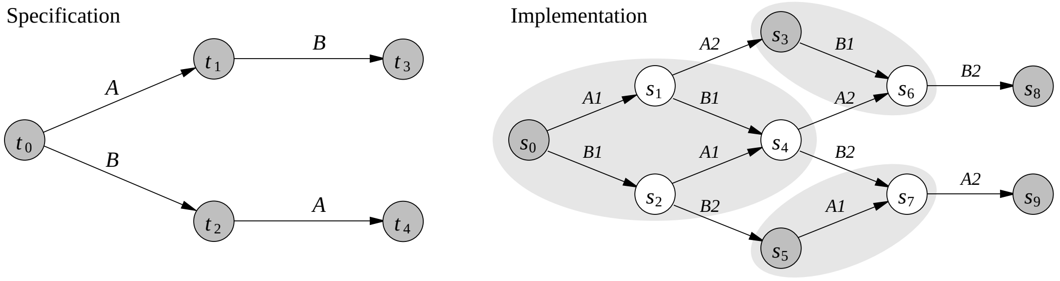


Figure 2.11: Backward Draining

to different results. Needless to say, s_0, s_3, s_5, s_8 and s_9 are drained terms that correspond to t_0, t_1, t_2, t_3 and t_4, respectively. Since forward draining would lead s_4 non-deterministically to s_8 or s_9, we use backward draining to cancel partially executed operations. Therefore, s_1, s_2 and s_4 are drained to s_0, s_6 is drained to s_3, and s_7 is drained to s_5.

We can use forward and backward draining simultaneously in order to complete some partially executed operations but cancel the others. For example, Figure 2.12 shows two different draining methods for the system given in Figure 2.10. The first method uses forward draining for rule A and backward draining for rule B, and the second method uses backward draining for rule B and forward draining for rule A. Note that proper combination of forward and backward draining effectively allows a term to be drained to any term, since it can always be rewritten to the initial term via backward draining and then to the desirable term via forward draining.

Forward draining can be achieved by selecting certain implementation rules as the draining rules. In contrast, backward draining may require that proper new rules be devised so that partially executed operations can be rolled back. The complication of backward draining is that certain implementation rules need to be augmented with hidden states to record the lost information necessary for backward draining. This can happen, for example, when more than one term can be written to the same term, or when the left-hand-side of a rule contains some variables that are not present in the right-hand-side of the rule. Generally speaking, backward draining is used when forward draining would lead to non-confluent results.

The formulation of mapping functions using drained terms is quite general. For example, we can compare a system with caches to a specification without caches by flushing all the cache cells. Similarly, we can compare a system with network queues to a specification without network queues by extracting all the messages from the queues. The idea of a rewriting sequence that can take a system into a drained term has an intuitive appeal for the system designer.

A typical soundness proof consists of the following steps:

- Specify a set of draining rules. Forward draining uses a subset of existing implementation rules, and backward draining may require new rules to be introduced. Forward and backward draining can be used together if necessary.

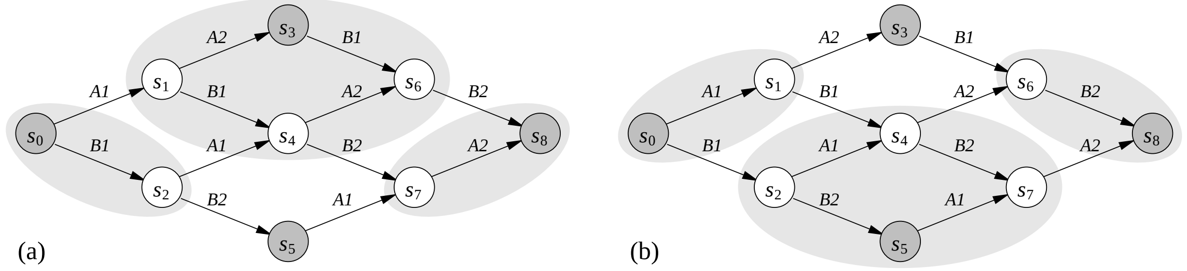


Figure 2.12: Combination of Forward and Backward Draining

- Show that the draining rules are strongly terminating and confluent, that is, rewriting a term with respect to the draining rules always terminates and reachs the same normal form, regardless of the order in which the rules are applied. This ensures the existence of a unique drained term for each implementation term.
- Define a mapping function that maps an implementation term to a specification term. The mapping function can often be specified as a projection function that removes unnecessary states from drained terms. Show that the mapping function maps the initial implementation term to the initial specification term.
- Prove that the specification can simulate the implementation with respect to the mapping function. That is, if s_1 can be rewritten to s_2 in the implementation, then the corresponding term of s_1 can be rewritten to the corresponding term of s_2 in the specification.

2.5.2 Verification of Liveness

We use temporal logic to reason about time-varying behaviors and liveness properties. An execution of a system can be described as a sequence of rewriting steps, each producing a new term by applying a rewriting rule on the current term. A sequence σ is a sequence of terms $\langle s_1, s_2, s_3, \ldots\rangle$ where s_1 is a legal term (that is, $s_0 \twoheadrightarrow s_1$ where s_0 is the initial term) and $s_i \rightarrow s_{i+1}$ (for $i = 1, 2, \ldots$).

A predicate can be defined using boolean operators and the temporal operator "$\Box$" (always). It can be a predicate for terms which contains no temporal operator, or a predicate for sequences of terms which contains some temporal operators. We say a sequence satisfies a term predicate if the first term of the sequence satisfies the predicate, and a sequence satisfies a rewriting rule if the first term of the sequence can be rewritten to the second term of the sequence according to the rule. Since all the boolean operators can be defined in terms of "$\neg$" and "$\wedge$", it suffices to define the semantics of predicates as follows:

$$\begin{aligned}
(\neg P)(\sigma) &\equiv \neg P(\sigma) \\
(P \wedge Q)(\sigma) &\equiv P(\sigma) \wedge Q(\sigma) \\
(\Box P)(\sigma) &\equiv \forall i \in \{1, 2, \ldots\}\ P(\langle s_i,\ s_{i+1},\ s_{i+2},\ \ldots\rangle)
\end{aligned}$$

Intuitively, "$\Box P$" means that "P is true all the times". We can define temporal operators such as "$\Diamond$" (eventually) and "$\leadsto$" (leads-to) using the operator "$\Box$".

- $\Diamond P \equiv \neg \Box \neg P$. This predicate asserts that P will be true at some future time (that is, P is not always false). Note that by "future time" we also include the "present time".
- $P \leadsto Q \equiv \Box(P \Rightarrow \Diamond Q)$. This predicate asserts that whenever P is true, Q will be true at some later time. This operator is transitive, meaning that any sequence satisfying "$P \leadsto Q$" and "$Q \leadsto G$" also satisfies "$P \leadsto G$".

To ensure liveness, we need to enforce some fairness throughout the system execution. Intuitively, fairness means that if a rule is applicable, it must be applied eventually. The fairness of concurrent systems can be expressed in terms of weak fairness and strong fairness conditions. Weak fairness means that if a rule is applicable, it must be applied eventually or will become impossible to apply at some later time. Strong fairness means that if a rule is applicable, it must be applied eventually or will become impossible to apply forever.

Let Enabled(R) be the predicate that determines whether the rule R is applicable. We can define weak fairness and strong fairness as follows:

$$\begin{array}{lcl} \mathrm{WF}(R) & \equiv & (\Box\Diamond R) \vee (\Box\Diamond\neg\mathrm{Enabled}(R)) \\ \mathrm{SF}(R) & \equiv & (\Box\Diamond R) \vee (\Diamond\Box\neg\mathrm{Enabled}(R)) \end{array}$$

The fairness of a rule actually refers to the fairness of the application of the rule on a specific redex. Note that a rule can often be applied on different redexes at the same time. Unless otherwise specified, weak or strong fairness of a rule means the application of the rule on any redex is weakly or strongly fair.

Theorem-WF Given predicates P and Q, $P \leadsto Q$ if the following conditions are true:

- $\forall s\,(P(s) \Rightarrow \exists R\,\exists s'\,(\mathrm{WF}(R) \wedge s \xrightarrow{R} s' \wedge Q(s')))$;
- $\forall s\,(P(s) \Rightarrow \forall R\,\forall s'\,(s \xrightarrow{R} s' \Rightarrow (Q(s') \vee P(s'))))$.

Theorem-SF Given predicates P and Q, $P \leadsto Q$ if the following conditions are true:

- $\forall s\,(P(s) \Rightarrow \exists R\,\exists s'\,(\mathrm{SF}(R) \wedge s \xrightarrow{R} s' \wedge Q(s')))$;
- $\forall s\,(P(s) \Rightarrow \forall R\,\forall s'\,(s \xrightarrow{R} s' \Rightarrow (Q(s') \vee \exists G\;(G(s') \wedge (G \leadsto P)))))$.

Although the operator "$\Box$" can be used to specify properties such as what will happen eventually, it does not allow us to express properties such as what will happen in the next term of a sequence. This constraint is deliberate because an implementation can often take various number of steps to achieve the semantic effect of a single step in a specification. Therefore, it makes little sense for a specification to require that something must happen in the next step of an execution.

However, some additional temporal operators can be used to simplify the liveness proof. The operator "$\circ$" (next) is introduced for this purpose: the predicate $(\circ P)(\langle s_1,\, s_2,\, s_3,\, \ldots\rangle)$

is true if and only if $P(\langle s_2,\, s_3,\, \ldots\rangle)$ is true. Intuitively, "$\circ P$" means that "P will be true in the next term of the sequence. Note that this operator cannot appear in liveness specifications; it can only be used in liveness proofs. In the following theorems, let P, Q, F, G and H be predicates regarding a sequence of terms.

Theorem-A: $P \leadsto Q$ if

- $(P \wedge G) \leadsto Q$;
- $\Box(P \Rightarrow \neg Q)$;
- $\Box((\neg P \wedge \circ P) \Rightarrow \circ G)$;
- $P(s_0) \Rightarrow G(s_0)$ where s_0 is the initial term of the system.

Proof Let σ be a sequence of terms $\langle s_0,\, s_1,\, s_2,\, \ldots\rangle$ where s_0 is the initial term. Without losing generality, we assume $\exists i \in \{0,1,\ldots\}\ P(s_i)$. There are two cases:

- $\forall j \in \{0,1,\ldots,i\}\ P(s_j)$. This implies $G(s_0)$ and $\forall j \in \{0,1,\ldots,i\}\ \neg Q(s_j)$. Therefore, $\exists k \in \{i+1, i+2, \ldots\}\ Q(s_k)$.
- $\exists m \in \{0,1,\ldots,i-1\}\ (\neg P(s_m) \wedge \forall j \in \{m+1,\ldots,i\}\ P(s_j))$. This implies $G(s_{m+1})$ and $\forall j \in \{m+1, m+2, \ldots, i\}\ \neg Q(s_j)$. Therefore, $\exists k \in \{i+1, i+2, \ldots\}\ Q(s_k)$. □

Theorem-B: $P \leadsto (P \wedge Q)$ if

- $P \leadsto Q$;
- $\Box((P \wedge \circ\neg P) \Rightarrow Q)$.

Proof Let σ be a sequence of terms $\langle s_1,\, s_2,\, \ldots\rangle$. Without losing generality, we assume $P(s_1)$. This implies $\exists i \in \{1,2,\ldots\}\ Q(s_i)$. There are two cases:

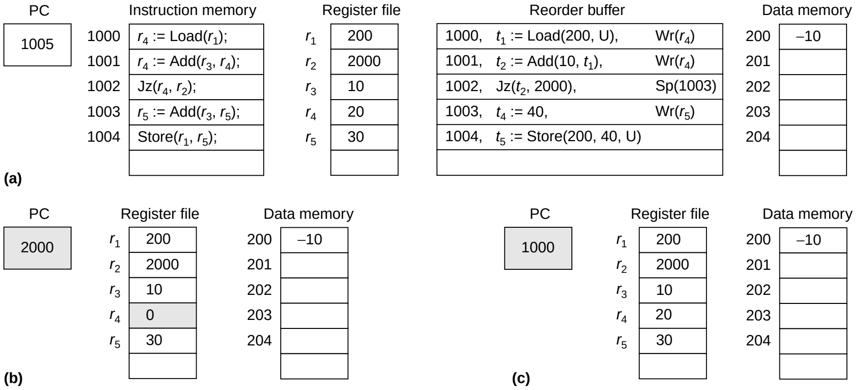
- $\forall j \in \{1,2,\ldots,i\}\ P(s_j)$. Therefore, $P(s_i) \wedge Q(s_i)$.
- $\exists k \in \{1,2,\ldots,i-1\}\ (P(s_k) \wedge \neg P(s_{k+1}))$. This implies $P(s_k) \wedge Q(s_k)$. □

Theorem-C: $(P \wedge G \wedge F) \leadsto ((Q \wedge G \wedge F) \vee H)$ if

- $P \leadsto Q$;
- $\Box((G \wedge \circ\neg G) \Rightarrow \circ H)$;
- $\Box((G \wedge F) \Rightarrow \circ F)$.

Proof Let σ be a sequence of terms $\langle s_1,\, s_2,\, \ldots\rangle$. Without losing generality, we assume $P(s_1) \wedge G(s_1) \wedge F(s_1)$. This implies $\exists i \in \{1,2,\ldots\}\ Q(s_i)$. There are two cases:

- $\forall j \in \{1,2,\ldots,i\}\ G(s_j)$. This implies $\forall j \in \{1,2,\ldots,i\}\ F(s_j)$. Therefore, $Q(s_i) \wedge G(s_i) \wedge F(s_i)$.
- $\exists k \in \{1,2,\ldots,i-1\}\ (G(s_k) \wedge \neg G(s_{k+1}))$. This implies $H(s_{k+1})$. □



(a)

PC: 1005

	Instruction memory
1000	r_4 := Load(r_1);
1001	r_4 := Add(r_3, r_4);
1002	Jz(r_4, r_2);
1003	r_5 := Add(r_3, r_5);
1004	Store(r_1, r_5);

	Register file
r_1	200
r_2	2000
r_3	10
r_4	20
r_5	30

Reorder buffer	
1000, t_1 := Load(200, U),	Wr(r_4)
1001, t_2 := Add(10, t_1),	Wr(r_4)
1002, Jz(t_2, 2000),	Sp(1003)
1003, t_4 := 40,	Wr(r_5)
1004, t_5 := Store(200, 40, U)	

	Data memory
200	−10
201	
202	
203	
204	

(b)

PC: 2000

	Register file
r_1	200
r_2	2000
r_3	10
r_4	0
r_5	30

	Data memory
200	−10
201	
202	
203	
204	

(c)

PC: 1000

	Register file
r_1	200
r_2	2000
r_3	10
r_4	20
r_5	30

	Data memory
200	−10
201	
202	
203	
204	

Figure 2.13: Draining the Processor

Throughout the thesis, the theorems above will be used in the proofs of the liveness of cache coherence protocols. General information about temporal and modal logic can be found elsewhere [40].

2.6 The Correctness of the $\mathrm{P_S}$ Model

One way to prove that the speculative processor is a correct implementation of the AX instruction set is to show that $\mathrm{P_B}$ and $\mathrm{P_S}$ can simulate each other in regard to some observable property. A natural observation function is the one that can extract all the programmer visible state, including the program counter, the register file and the memory, from the system. One can think of an observation function in terms of a print instruction that prints part or all of the programmer visible state. If model A can simulate model B, then for any program, model A should be able to print whatever model B prints during the execution.

The programmer visible state of $\mathrm{P_B}$ is obvious – it is the whole term. The $\mathrm{P_B}$ model does not have any hidden state. It is a bit tricky to extract the corresponding values of *pc*, *rf* and *mem* from the $\mathrm{P_S}$ model because of the partially or speculatively executed instructions. However, if we consider only those $\mathrm{P_S}$ states where the *rob*, *pmb* and *mpb* are empty, then it is straightforward to find the corresponding $\mathrm{P_B}$ state. We will call such states of $\mathrm{P_S}$ the *drained states*.

It is easy to show that $\mathrm{P_S}$ can simulate each rule of $\mathrm{P_B}$. Given a $\mathrm{P_B}$ term s_1, a $\mathrm{P_S}$ term t_1 is created such that it has the same values of *pc*, *rf*, *im* and *mem*, and its *rob*, *pmb* and *mpb* are all empty. Now, if s_1 can be rewritten to s_2 according to some $\mathrm{P_B}$ rule, we can apply a sequence of $\mathrm{P_S}$ rules to t_1 to obtain t_2 such that t_2 is in a drained state and has the same programmer visible state as s_2. In this manner, $\mathrm{P_S}$ can simulate each move of $\mathrm{P_B}$.

Simulation in the other direction is tricky because we need to find a $\mathrm{P_B}$ term corresponding to each term of $\mathrm{P_S}$ (not just the terms in the drained state). We need to somehow extract the programmer visible state from any $\mathrm{P_S}$ term. There are several ways to drive a $\mathrm{P_S}$ term to a

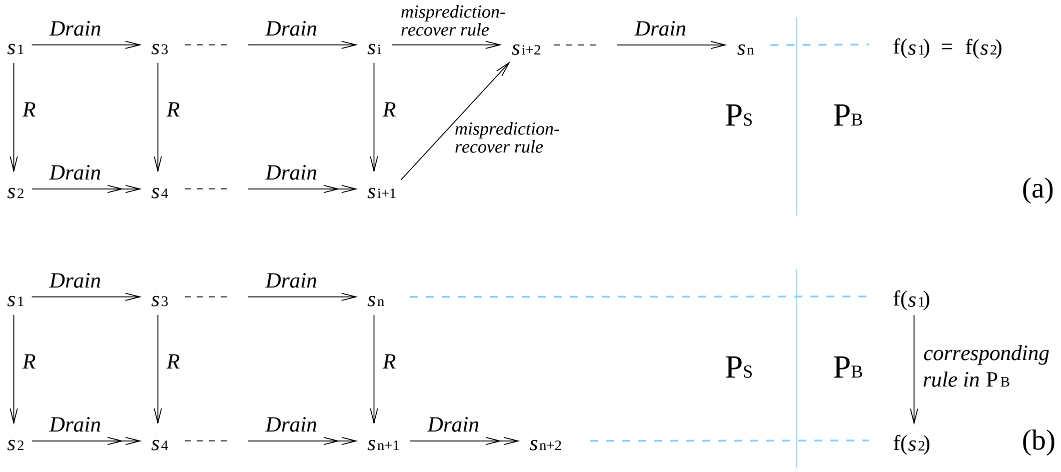


Figure 2.14: Simulation of Instruction Fetch Rules

drained state using the P_{S} rules, and each way may lead to a different drained state.

As an example, consider the snapshot shown in Figure 2.13 (a) (we have not shown *pmb* and *mpb*; let us assume both are empty). There are at least two ways to drive this term into a drained state. One way is to stop fetching instructions and complete all the partially executed instructions. This process can be thought of as applying a subset of the P_{S} rules (all the rules except the instruction fetch rules) to the term. After repeated application of such rules, the *rob* should become empty and the system should reach a drained state. Figure 2.13 (b) shows such a situation, where in the process of draining the pipeline, we discover that the branch speculation was wrong. An alternative way is to rollback the execution by killing all the partially executed instructions and restoring the *pc* to the address of the oldest killed instruction. Figure 2.13 (c) shows that the drained state obtained in this manner. Note that this drained state is different from the one obtained by completing the partially executed instructions. The two draining methods represent two extremes. By carefully selecting the rules that are applied to reach the drained state, we can allow certain instructions in the *rob* to be completed and the rest to be killed.

In the remainder of this section, we demonstrate the simulation of P_{S} by P_{B} with respect to the draining rules that include all the P_{S} rules except the instruction fetch rules. It can be shown by induction and case analysis that the draining rules are strongly terminating and confluent, that is, rewriting a P_{S} term with respect to the draining rules always terminates and reaches the same normal form regardless of the order in which the rules are applied. Furthermore, it can be shown that the *rob*, *pmb* and *mpb* all become empty in the normal form, which is also referred to as the drained term. Given a P_{S} term s, the corresponding P_{B} term $\text{f}(s)$ can be obtained by projecting the drained term to a P_{B} term (that is, removing *rob*, *btb*, *pmb* and *mpb*).

It is obvious that the initial P_{S} term is mapped to the initial P_{B} term. Assume s_1 can be rewritten to s_2 according to rule R. If R is not an instruction fetch rule, then s_1 and s_2 have the same drained term since R is a draining rule. Figure 2.14 shows the simulation when R is an instruction fetch rule. We use s_{n} to represent the drained term of s_1. There are two possible cases:

- If the draining of s_1 ever invokes a misprediction-recover rule (*Jump-WrongSpec* or *NoJump-WrongSpec*), then s_1 and s_2 have the same drained term (see Figure 2.14 (a)).
- If the draining of s_1 does not invoke any misprediction-recover rule, then there exists a term $s_{\mathrm{n+1}}$ such that s_{n} can be rewritten to $s_{\mathrm{n+1}}$ according to rule R, while s_2 can be rewritten to $s_{\mathrm{n+1}}$ according to the draining rules. Therefore, $\mathrm{f}(s_1)$ can be rewritten to $\mathrm{f}(s_2)$ according to the corresponding $\mathrm{P_B}$ rule (see Figure 2.14 (b)).

When a system has many rules, the correctness proofs can quickly become tedious. Use of theorem provers, such as PVS [95], and model checkers, such as Murphi [35], can alleviate this problem. TRS descriptions, augmented with proper information about the system building blocks, also hold the promise of high-level synthesis. High-level architectural descriptions that are both automatically synthesizable and verifiable would permit architectural exploration at a fraction of the time and cost required by current commercial tools.

2.7 Relaxed Memory Access Operations

Memory access instructions can be implemented more aggressively while still preserving the instruction semantics. For example, write buffers and non-blocking caches can be employed to reduce or hide memory access latencies. We can use relaxed memory access rules to model these architectural optimizations. Such techniques are often aimed at performance optimization for sequential programs, and can have subtle semantic implications in multiprocessor systems. They are transparent for sequential programs but often exposed programmatically for parallel programs.

Write buffers: A common architectural optimization is to allow a load instruction to bypass preceding store instructions which are waiting in the write buffer, as long as there is no address conflict between the load and store instructions.

$$\begin{array}{ll} & \mathrm{Sys}(proc,\ pmb_1;\langle t,\mathrm{Load}(a)\rangle;pmb_2,\ mpb,\ mem) \qquad \textit{if}\quad \mathrm{Load},\ \mathrm{Store}(a,\text{-}) \notin pmb_1 \\ \rightarrow & \mathrm{Sys}(proc,\ pmb_1;pmb_2,\ mpb|\langle t,mem[a]\rangle,\ mem) \end{array}$$

Short-circuiting: It is common for an architecture with write buffers to allow a load instruction to obtain the data from the write buffer if there is an outstanding store instruction to the same address.

$$\begin{array}{ll} & \mathrm{Sys}(proc,\ pmb_1;\langle t',\mathrm{Store}(a,v)\rangle;pmb_2;\langle t,\mathrm{Load}(a)\rangle;pmb_3,\ mpb,\ mem) \qquad \textit{if}\quad \mathrm{Store}(a,\text{-}) \notin pmb_2 \\ \rightarrow & \mathrm{Sys}(proc,\ pmb_1;\langle t',\mathrm{Store}(a,v)\rangle;pmb_2;pmb_3,\ mpb|\langle t,v\rangle,\ mem) \end{array}$$

Non-FIFO write buffers: Some architectures with write buffers assume that store instructions can be reordered in a write buffer. This allows stores to the same cache line to be merged into one burst bus transactions.

$$\begin{array}{ll} & \text{Sys}(\mathit{proc},\ \mathit{pmb}_1;\langle t,\text{Store}(a,v)\rangle;\mathit{pmb}_2,\ \mathit{mpb},\ \mathit{mem}) \qquad \textit{if}\quad \text{Load},\ \text{Store}(a,\text{-}) \notin \mathit{pmb}_1 \\ \rightarrow & \text{Sys}(\mathit{proc},\ \mathit{pmb}_1;\mathit{pmb}_2,\ \mathit{mpb}|\langle t,\text{Ack}\rangle,\ \mathit{mem}[a{:=}v]) \end{array}$$

Non-blocking caches: Modern architectures often have non-blocking caches to allow multiple outstanding load instructions. This effectively allows loads to be performed out-of-order.

$$\begin{array}{ll} & \text{Sys}(\mathit{proc},\ \mathit{pmb}_1;\langle t,\text{Load}(a)\rangle;\mathit{pmb}_2,\ \mathit{mpb},\ \mathit{mem}) \qquad \textit{if}\quad \text{Store} \notin \mathit{pmb}_1 \\ \rightarrow & \text{Sys}(\mathit{proc},\ \mathit{pmb}_1;\mathit{pmb}_2,\ \mathit{mpb}|\langle t,\mathit{mem}[a]\rangle,\ \mathit{mem}) \end{array}$$

A relaxed implementation of memory operations allows memory accesses to be performed in arbitrary order, provided that data dependences of the program are preserved. A load instruction can be performed if there is no preceding store instruction on the same address. A store instruction can be performed if there is no preceding load or store instruction on the same address. The short-circuit rule allows a load to obtain the data from a preceding store before the data is written to the memory.

Relaxed-Load-Memory Rule

$$\begin{array}{ll} & \text{Sys}(\mathit{proc},\ \mathit{pmb}_1;\langle t,\text{Load}(a)\rangle;\mathit{pmb}_2,\ \mathit{mpb},\ \mathit{mem}) \qquad \textit{if}\quad \text{Store}(a,\text{-}) \notin \mathit{pmb}_1 \\ \rightarrow & \text{Sys}(\mathit{proc},\ \mathit{pmb}_1;\mathit{pmb}_2,\ \mathit{mpb}|\langle t,\mathit{mem}[a]\rangle,\ \mathit{mem}) \end{array}$$

Relaxed-Store-Memory Rule

$$\begin{array}{ll} & \text{Sys}(\mathit{proc},\ \mathit{pmb}_1;\langle t,\text{Store}(a,v)\rangle;\mathit{pmb}_2,\ \mathit{mpb},\ \mathit{mem}) \qquad \textit{if}\quad \text{Load}(a),\ \text{Store}(a,\text{-}) \notin \mathit{pmb}_1 \\ \rightarrow & \text{Sys}(\mathit{proc},\ \mathit{pmb}_1;\mathit{pmb}_2,\ \mathit{mpb}|\langle t,\text{Ack}\rangle,\ \mathit{mem}[a{:=}v]) \end{array}$$

Short-Circuit Rule

$$\begin{array}{ll} & \text{Sys}(\mathit{proc},\ \mathit{pmb}_1;\langle t',\text{Store}(a,v)\rangle;\mathit{pmb}_2;\langle t,\text{Load}(a)\rangle;\mathit{pmb}_3,\ \mathit{mpb},\ \mathit{mem}) \qquad \textit{if}\quad \text{Store}(a,\text{-}) \notin \mathit{pmb}_2 \\ \rightarrow & \text{Sys}(\mathit{proc},\ \mathit{pmb}_1;\langle t',\text{Store}(a,v)\rangle;\mathit{pmb}_2;\mathit{pmb}_3,\ \mathit{mpb}|\langle t,v\rangle,\ \mathit{mem}) \end{array}$$

We can use the relaxed memory access rules above as a more efficient processor-memory interface for P_S (this inevitably leads to a relaxed memory model in multiprocessor systems). We can further allow memory instructions to be dispatched out-of-order (t' represents an unresolved tag):

Relaxed-Dispatch-Load Rule

$$\begin{array}{ll} & \text{Sys}(\text{Proc}(\mathit{ia},\ \mathit{rf},\ \mathit{rob}_1 \oplus \text{Itb}(\mathit{ia}_1,t{:=}\text{Load}(a,\text{U}),\text{Wr}(r)) \oplus \mathit{rob}_2,\ \mathit{btb},\ \mathit{im}),\ \mathit{pmb},\ \mathit{mpb},\ \mathit{mem}) \\ & \qquad\qquad \textit{if}\quad \text{Jz},\ \text{Load}(t',\text{U}),\ \text{Store}(\text{-},t',\text{U}),\ \text{Store}(t',\text{-},\text{U})\ \text{Store}(a,\text{-},\text{U}) \notin \mathit{rob}_1 \\ \rightarrow & \text{Sys}(\text{Proc}(\mathit{ia},\ \mathit{rf},\ \mathit{rob}_1 \oplus \text{Itb}(\mathit{ia}_1,t{:=}\text{Load}(a,\text{D}),\text{Wr}(r)) \oplus \mathit{rob}_2,\ \mathit{btb},\ \mathit{im}),\ \mathit{pmb};\langle t,\text{Load}(a)\rangle,\ \mathit{mpb}, \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{mem}) \end{array}$$

Relaxed-Dispatch-Store Rule

$$\begin{array}{ll} & \text{Sys}(\text{Proc}(\mathit{ia},\ \mathit{rf},\ \mathit{rob}_1 \oplus \text{Itb}(\mathit{ia}_1,t{:=}\text{Store}(a,v,\text{U})) \oplus \mathit{rob}_2,\ \mathit{btb},\ \mathit{im}),\ \mathit{pmb},\ \mathit{mpb},\ \mathit{mem}) \\ & \qquad\qquad \textit{if}\quad \text{Jz},\ \text{Load}(t',\text{U}),\ \text{Store}(\text{-},t',\text{U}),\ \text{Store}(t',\text{-},\text{U})\ \text{Load}(a,\text{U}),\ \text{Store}(a,\text{-},\text{U}) \notin \mathit{rob}_1 \\ \rightarrow & \text{Sys}(\text{Proc}(\mathit{ia},\ \mathit{rf},\ \mathit{rob}_1 \oplus \text{Itb}(\mathit{ia}_1,t{:=}\text{Store}(a,v,\text{D})) \oplus \mathit{rob}_2,\ \mathit{btb},\ \mathit{im}),\ \mathit{pmb};\langle t,\text{Store}(a,v)\rangle,\ \mathit{mpb},\ \mathit{mem}) \end{array}$$

It can be shown that the relaxed dispatch rules have no semantic impact on program behaviors in multiprocessor systems that employ the relaxed memory access rules. This is obvious because a processor dispatches a memory instruction only when there is no data dependence between the instruction and all the preceding instructions which have not been dispatched.

The relaxed dispatch rules do not allow a memory instruction to be dispatched if a preceding instruction contains an unresolved tag. A more aggressive implementation can relax this constraint so that a memory instruction can be dispatched even if a preceding instruction contains an unknown address or data, as long as there is no data dependence. Furthermore, we can allow speculative load instructions to be dispatched. (This requires slight modification of the branch completion rules since a dispatched load instruction on a wrong speculation path cannot be killed before the response from the memory is discarded). Therefore, a load instruction can be dispatched if there is no undispatched store in front it in the *rob* that may write to the same address; a store instruction can be dispatched if it is not on a speculative path and there is no undispatched load or store in front it in the *rob* that may read from or write to the same address.

Aggressive-Dispatch-Load Rule

$\text{Sys}(\text{Proc}(ia,\ rf,\ rob_1 \oplus \text{Itb}(ia_1, t{:=}\text{Load}(a,\text{U}),\text{Wr}(r)) \oplus rob_2,\ btb,\ im),\ pmb,\ mpb,\ mem)$

$\quad\quad \textit{if} \quad \text{Store}(a,\text{-},\text{U}),\ \text{Store}(t',\text{-},\text{U}) \notin rob_1$

$\rightarrow \quad \text{Sys}(\text{Proc}(ia,\ rf,\ rob_1 \oplus \text{Itb}(ia_1, t{:=}\text{Load}(a,\text{D}),\text{Wr}(r)) \oplus rob_2,\ btb,\ im),\ pmb;\langle t,\text{Load}(a)\rangle,\ mpb,\ mem)$

Aggressive-Dispatch-Store Rule

$\text{Sys}(\text{Proc}(ia,\ rf,\ rob_1 \oplus \text{Itb}(ia_1, t{:=}\text{Store}(a,v,\text{U})) \oplus rob_2,\ btb,\ im),\ pmb,\ mpb,\ mem)$

$\quad\quad \textit{if} \quad \text{Jz},\ \text{Load}(a,\text{U}),\ \text{Load}(t',\text{U}),\ \text{Store}(a,\text{-},\text{U}),\ \text{Store}(t',\text{-},\text{U}) \notin rob_1$

$\rightarrow \quad \text{Sys}(\text{Proc}(ia,\ rf,\ rob_1 \oplus \text{Itb}(ia_1, t{:=}\text{Store}(a,v,\text{D})) \oplus rob_2,\ btb,\ im),\ pmb;\langle t,\text{Store}(a,v)\rangle,\ mpb,\ mem)$

In the architectural optimizations we have discussed so far, we do not allow value speculation [82, 83]. A memory instruction cannot be performed if it has an unknown address or data. Furthermore, a load instruction cannot be performed if a preceding store instruction has an unresolved address, and a store instruction cannot be performed if a preceding load or store instruction has an unresolved address. We also do not allow speculative store instructions. Such constraints can be relaxed with value speculation to achieve better performance. Conceptually, value speculation allows a processor to chose an arbitrary value for any unresolved variable at any time, provided that all the speculative instructions can be killed if the speculation turns out to be wrong. Many speculative mechanisms and compiler optimizations are special cases of value speculation. However, the impact of value speculation on program behaviors can be difficult to understand in multiprocessor systems.

Modern microprocessors contain various architecture features designed to improve performance, such as branch prediction, speculative execution, non-blocking caches, dynamic scheduling, superscalar execution and many others. All these features add complexity to the architecture and interact with each other, making the verification task more difficult. In recent years, considerable attention has focused on formal specification and verification of architectures [22, 23, 32, 57, 88, 104, 123]. For example, Burch and Dill [23] described a technique which automatically compares a pipelined implementation to an architectural specification and produces debugging information for incorrect processor design. Levitt and Olukotun [80] proposed a methodology that iteratively de-constructs a pipeline by merging adjacent pipeline stages thereby allowing verifications to be done in a number of easier steps. Windley [123] presented a case study which uses abstract theories to hierarchically verify microprocessor implementations formalized in HOL. Windley's methodology is similar to ours in the sense that the correctness theorem states the implementation implies the behavior specification. The most critical step in the proof is the definition of the abstract mapping function to map the states of the implementation system into the states of the specification system. Our mapping functions based on draining rules are more intuitive because of the use of TRSs.

Chapter 3

The Commit-Reconcile & Fences Memory Model

CRF (Commit-Reconcile & Fences) is a mechanism-oriented memory model and intended for architects and compiler writers rather than for high-level parallel programming. It is defined by giving algebraic semantics to the memory related instructions so that every CRF program has a well-defined operational behavior. The CRF mechanisms give architects great flexibility for efficient implementations, and give compiler writers all the control they need. They can be incorporated in stages in future systems without loss of compatibility with existing systems.

In Section 3.1 we define the CRF memory model using Term Rewriting Systems. We then present some derived rules of CRF in Section 3.2, and discuss coarse-grain CRF instructions in Section 3.3. Section 3.4 demonstrates the universality of CRF by showing that programs under many existing memory models can be translated into CRF programs, and CRF programs can run efficiently on many existing systems. A generalized model of CRF is presented in Section 3.5, which allows writeback operations to be performed in different orders with respect to different cache sites.

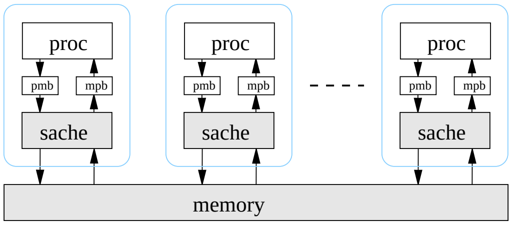
3.1 The CRF Model

The CRF model has a semantic notion of caches, referred to as *saches*, which makes the operational behavior of data replication to be part of the model. Figure 3.1 gives the CRF instructions. The CRF instructions can be classified into three categories: memory access instructions Loadl (load-local) and Storel (store-local), memory rendezvous instructions Commit and Reconcile, and memory fence instructions. There are four types of memory fences: Fence_{rr}, Fence_{rw}, Fence_{wr} and Fence_{ww}, which correspond to read-read, read-write, write-read, and write-write fences, respectively.

CRF exposes both data replication and instruction reordering at the Instruction Set Architecture (ISA) level. Figure 3.2 gives the system configuration of CRF. The system contains a memory and a set of sites. Each site has a semantic cache, on which Loadl and Storel instructions operate. The Commit and Reconcile instructions can be used to ensure that the data

$$\begin{array}{lcl}
\mathsf{INST} & \equiv & \mathrm{Loadl}(a) \;\; [\!] \;\; \mathrm{Storel}(a,v) \\
 & [\!] & \mathrm{Commit}(a) \;\; [\!] \;\; \mathrm{Reconcile}(a) \\
 & [\!] & \mathrm{Fence}_{\mathrm{rr}}(a_1,a_2) \;\; [\!] \;\; \mathrm{Fence}_{\mathrm{rw}}(a_1,a_2) \\
 & [\!] & \mathrm{Fence}_{\mathrm{wr}}(a_1,a_2) \;\; [\!] \;\; \mathrm{Fence}_{\mathrm{ww}}(a_1,a_2)
\end{array}$$

Figure 3.1: CRF Instructions



SYS	$\equiv$	Sys(MEM, SITEs)	*System*
SITEs	$\equiv$	SITE $[\!]$ SITE | SITEs	*Set of Sites*
SITE	$\equiv$	Site(SACHE, PMB, MPB, PROC)	*Site*
SACHE	$\equiv$	ϵ $[\!]$ Cell(a,v,CSTATE) | SACHE	*Semantic Cache*
CSTATE	$\equiv$	Clean $[\!]$ Dirty	*Cache State*
PMB	$\equiv$	ϵ $[\!]$ $\langle t$,INST$\rangle$;PMB	*Processor-to-Memory Buffer*
MPB	$\equiv$	ϵ $[\!]$ $\langle t$,REPLY$\rangle$|MPB	*Memory-to-Processor Buffer*
REPLY	$\equiv$	v $[\!]$ Ack	*Reply*

Figure 3.2: System Configuration of CRF

produced by one processor can be observed by another processor whenever necessary. CRF allows memory accesses to be reordered as long as data dependence constraints are preserved.

The definition of CRF includes two sets of rules. The first set of rules specifies the execution of Loadl, Storel, Commit and Reconcile instructions. It also includes rules that govern the data propagation between semantic caches and memory. The second set of rules deals with instruction reordering and memory fences. We also refer to the first set of rules as the Commit-Reconcile (CR) model, because these rules by themselves define a memory model which is the same as CRF except that instructions are executed strictly in-order.

3.1.1 The CR Model

There are two states for sache cells, Clean and Dirty. The Clean state indicates that the data has not been modified since it was cached or last written back. The Dirty state indicates that the data has been modified and has not been written back to the memory since then. Note in CRF, different saches can have a cell with the same address but different values.

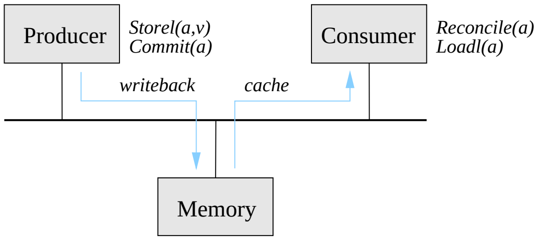


Figure 3.3: Producer-Consumer Synchronization in CRF

Loadl and Storel Rules: A Loadl or Storel can be performed if the address is cached in the sache.

CRF-Loadl Rule

$\quad \text{Site}(sache,\ \langle t,\text{Loadl}(a)\rangle;pmb,\ mpb,\ proc) \qquad \textit{if} \quad \text{Cell}(a,v,\text{-}) \in sache$

$\rightarrow \quad \text{Site}(sache,\ pmb,\ mpb|\langle t,v\rangle,\ proc)$

CRF-Storel Rule

$\quad \text{Site}(\text{Cell}(a,\text{-},\text{-}) \mid sache,\ \langle t,\text{Storel}(a,v)\rangle;pmb,\ mpb,\ proc)$

$\rightarrow \quad \text{Site}(\text{Cell}(a,v,\text{Dirty}) \mid sache,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc)$

Although the store rule above requires that the address be cached before the Storel can be performed, it makes no semantic difference to allow the Storel to be performed even if the address is not cached. This is because, if the address is not cached, the sache can first obtain a Clean copy from the memory (by applying the cache rule given below), and then perform the Storel access. This can be represented by a straightforward derived rule.

Commit and Reconcile Rules: A Commit can be completed if the address is uncached or cached in the Clean state. A Reconcile can be completed if the address is uncached or cached in the Dirty state.

CRF-Commit Rule

$\quad \text{Site}(sache,\ \langle t,\text{Commit}(a)\rangle;pmb,\ mpb,\ proc) \qquad \textit{if} \quad \text{Cell}(a,\text{-},\text{Dirty}) \notin sache$

$\rightarrow \quad \text{Site}(sache,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc)$

CRF-Reconcile Rule

$\quad \text{Site}(sache,\ \langle t,\text{Reconcile}(a)\rangle;pmb,\ mpb,\ proc) \qquad \textit{if} \quad \text{Cell}(a,\text{-},\text{Clean}) \notin sache$

$\rightarrow \quad \text{Site}(sache,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc)$

On a Commit operation, if the address is cached and the cell's state is Dirty, the data must be first written back to the memory (by applying the writeback rule given below). On a Reconcile operation, if the address is cached and the cell's state is Clean, the cell must be first purged from the sache (by applying the purge rule given below).

We can use Commit and Reconcile instructions to implement producer-consumer synchronization. The memory behaves as the rendezvous between the producer and the consumer: the

Processor Rules				
Rule Name	**Instruction**	**Cstate**	**Action**	**Next Cstate**
CRF-Loadl	Loadl(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)
		Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)
CRF-Storel	Storel(a,v)	Cell(a,-,Clean)	*retire*	Cell(a,v,Dirty)
		Cell(a,-,Dirty)	*retire*	Cell(a,v,Dirty)
CRF-Commit	Commit(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)
		$a \notin sache$	*retire*	$a \notin sache$
CRF-Reconcile	Reconcile(a)	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)
		$a \notin sache$	*retire*	$a \notin sache$

Background Rules				
Rule Name	**Cstate**	**Mstate**	**Next Cstate**	**Next Mstate**
CRF-Cache	$a \notin sache$	Cell(a,v)	Cell(a,v,Clean)	Cell(a,v)
CRF-Writeback	Cell(a,v,Dirty)	Cell(a,-)	Cell(a,v,Clean)	Cell(a,v)
CRF-Purge	Cell(a,-,Clean)	Cell(a,v)	$a \notin sache$	Cell(a,v)

Figure 3.4: Summary of CR Rules

producer performs a commit operation to guarantee that the modified data has been written back to the memory, while the consumer performs a reconcile operation to guarantee that the stale copy, if any, has been purged from the sache so that subsequent load operations must retrieve the data from the memory (see Figure 3.3).

Cache, Writeback and Purge Rules: A sache can obtain a Clean copy from the memory, if the address is not cached at the time (thus no sache can contain more than one copy for the same address). A Dirty copy can be written back to the memory, after which the sache state becomes Clean. A Clean copy can be purged from the sache at any time, but cannot be written back to the memory. These rules are also called background rules, since they can be applied even though no instruction is executed by any processor.

CRF-Cache Rule

$$\begin{array}{ll} & \text{Sys}(mem,\ \text{Site}(sache,\ pmb,\ mpb,\ proc)\ |\ sites) \qquad \textit{if} \quad a \notin sache \\ \rightarrow & \text{Sys}(mem,\ \text{Site}(\text{Cell}(a,mem[a],\text{Clean})\ |\ sache,\ pmb,\ mpb,\ proc)\ |\ sites) \end{array}$$

CRF-Writeback Rule

$$\begin{array}{ll} & \text{Sys}(mem,\ \text{Site}(\text{Cell}(a,v,\text{Dirty})\ |\ sache,\ pmb,\ mpb,\ proc)\ |\ sites) \\ \rightarrow & \text{Sys}(mem[a{:=}v],\ \text{Site}(\text{Cell}(a,v,\text{Clean})\ |\ sache,\ pmb,\ mpb,\ proc)\ |\ sites) \end{array}$$

CRF-Purge Rule

$$\begin{array}{ll} & \text{Site}(\text{Cell}(a,\text{-},\text{Clean})\ |\ sache,\ pmb,\ mpb,\ proc) \\ \rightarrow & \text{Site}(sache,\ pmb,\ mpb,\ proc) \end{array}$$

The CR rules are summarized in Figure 3.4. The seven rules are classified into two categories: the processor rules and the background rules. Each processor rule processes a memory instruction to completion. When an instruction is completed (retired), it is removed from the processor-to-memory buffer and the corresponding data or acknowledgment is sent to the

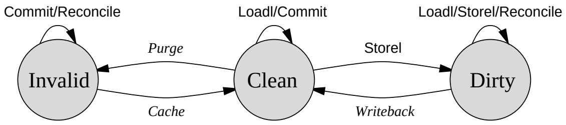


Figure 3.5: Semantic Cache State Transitions of CRF

memory-to-processor buffer. Each background rule involves state transitions at both the sache and the memory side. Figure 3.5 shows the sache state transitions of CRF.

3.1.2 Reordering and Fence Rules

CRF allows reordering of memory accesses provided data dependence constraints are preserved, and provides memory fences to enforce ordering if needed. We chose fine-grain fences to define the memory model. There are four types of memory fences to control reordering: Fence_{rr}, Fence_{rw}, Fence_{wr} and Fence_{ww}. Each memory fence has a pair of arguments, a pre-address and a post-address, and imposes an ordering constraint between memory operations involving the corresponding addresses. For example, $\text{Fence}_{\text{rw}}(a_1,a_2)$ ensures that any preceding Loadl to location a_1 must be performed before any following Storel to location a_2 can be performed. This implies that instructions Loadl(a_1) and Storel(a_2,v) separated by $\text{Fence}_{\text{rw}}(a_1,a_2)$ cannot be reordered.

To allow maximal reordering flexibility, CRF allows memory access instructions to be reordered if they access different addresses or if they are both Loadl instructions. Furthermore, memory rendezvous instructions can always be reordered with respect to each other, and memory fence instructions can always be reordered with respect to each other. CRF does not reorder the following instruction sequences:

- Storel(a,-); Loadl(a)
- Loadl(a); Storel(a,-)
- Storel(a,-); Storel(a,-)
- Reconcile(a); Loadl(a)
- Storel(a,-); Commit(a)
- Loadl(a); $\text{Fence}_{\text{r}\alpha}(a,\text{-})$
- Commit(a); $\text{Fence}_{\text{w}\alpha}(a,\text{-})$
- $\text{Fence}_{\alpha\text{r}}(a)$; Reconcile($a$)
- $\text{Fence}_{\alpha\text{w}}(a)$; Storel($a$,-)

We use shorthand notation $\text{Fence}_{\text{r}\alpha}$ to represent a Fence_{rr} or Fence_{rw} instruction, and $\text{Fence}_{\text{w}\alpha}$ to represent a Fence_{wr} or Fence_{ww} instruction. Similarly, we use $\text{Fence}_{\alpha\text{r}}$ to represent a Fence_{rr} or Fence_{wr} instruction, and $\text{Fence}_{\alpha\text{w}}$ to represent a Fence_{rw} or Fence_{ww} instruction. We use $\text{Fence}_{\alpha\beta}$ to represent any type of fence instruction. Note that a $\text{Fence}_{\text{w}\alpha}$ instruction

$\mathbf{I}_2 \Rightarrow$ $\mathbf{I}_1 \Downarrow$	Loadl (a')	Storel (a',v')	Fence$_{\text{rr}}$ (a'_1,a'_2)	Fence$_{\text{rw}}$ (a'_1,a'_2)	Fence$_{\text{wr}}$ (a'_1,a'_2)	Fence$_{\text{ww}}$ (a'_1,a'_2)	Commit (a')	Reconcile (a')
Loadl(a)	true	$a \neq a'$	$a \neq a'_1$	$a \neq a'_1$	true	true	true	true
Storel(a,v)	$a \neq a'$	$a \neq a'$	true	true	true	true	$a \neq a'$	true
Fence$_{\text{rr}}(a_1,a_2)$	true	true	true	true	true	true	true	$a_2 \neq a'$
Fence$_{\text{rw}}(a_1,a_2)$	true	$a_2 \neq a'$	true	true	true	true	true	true
Fence$_{\text{wr}}(a_1,a_2)$	true	true	true	true	true	true	true	$a_2 \neq a'$
Fence$_{\text{ww}}(a_1,a_2)$	true	$a_2 \neq a'$	true	true	true	true	true	true
Commit(a)	true	true	true	true	$a \neq a'_1$	$a \neq a'_1$	true	true
Reconcile(a)	$a \neq a'$	true	true	true	true	true	true	true

Figure 3.6: Instruction Reordering Table of CRF

imposes ordering constraints on preceding Commit instructions instead of Storel instructions, since only a Commit can force the dirty copy to be written back to the memory. It makes little sense to ensure a Storel is completed if it is not followed by a Commit. Likewise, a Fence$_{\alpha\text{r}}$ instruction imposes ordering constraints on following Reconcile instructions instead of Loadl instructions, since only a Reconcile can force a stale copy to be purged. It makes little sense to postpone a Loadl if it is not preceded by a Reconcile.

Figure 3.6 concisely defines the conditions under which two adjacent CRF instructions can be reordered (assume instruction I_1 precedes instruction I_2, and a 'true' condition indicates that the reordering is allowed). The underlying rationale is to allow maximum reordering flexibility for out-of-order execution. There are 64 reordering rules defined by the reordering table. As an example, the rule corresponding to the Storel-Storel entry allows two Storel instructions to be reordered if they access different addresses:

CRF-Reorder-Storel-Storel Rule

$$\begin{array}{ll} & \langle t,\text{Storel}(a,v)\rangle;\langle t',\text{Storel}(a',v')\rangle \qquad \textit{if} \quad a \neq a' \\ \rightarrow & \langle t',\text{Storel}(a',v')\rangle;\langle t,\text{Storel}(a,v)\rangle \end{array}$$

This rule is commutative in the sense that reordered instructions can be reordered back. However, not all the reordering rules commute. For example, the rule represented by the Fence$_{\text{rr}}$-Loadl entry does not commute: once the reordering is performed, the instructions cannot be reordered back unless the address of the Loadl instruction and the pre-address of the Fence$_{\text{rr}}$ instruction are different (according to the Loadl-Fence$_{\text{rr}}$ entry).

We also need a rule to discharge a memory fence:

CRF-Fence Rule

$$\begin{array}{ll} & \text{Site}(\textit{sache},\ \langle t,\text{Fence}_{\alpha\beta}(a_1,a_2)\rangle;\textit{pmb},\ \textit{mpb},\ \textit{proc}) \\ \rightarrow & \text{Site}(\textit{sache},\ \textit{pmb},\ \textit{mpb}|\langle t,\text{Ack}\rangle,\ \textit{proc}) \end{array}$$

It is worth pointing out that appropriate extension is needed for CRF in order to define the semantics of synchronization instructions such as Test-&-Set, Swap, Lock/Unlock and Load-&-Reserve/Store-Conditional. The CRF model by itself contains no particular synchronization instructions, because of the lack of consensus on what synchronization instructions should be supported. To capture the atomicity requirement of read-modify-write operations, we need to introduce a notion of atomic sequence, which implies that some sequence of operations must be performed atomically with respect to other operations.

We have defined the CRF memory model as a mathematical relation between a stream of memory instructions from the processor and a stream of legal responses from the memory. The precise definition can be used to verify the correctness of architecture optimizations and cache coherence protocols. However, given a program, its observable behavior on a computer can be affected both by the memory architecture such as caches and cache coherence protocols, and the processor architecture such as instruction reordering and speculative execution. For example, many programs may exhibit different behaviors if memory instructions are issued out-of-order. The memory instruction dispatch rules given in Sections 2.4 and 2.7 are examples of how memory instruction streams can be generated by a processor.

3.2 Some Derived Rules of CRF

A derived rule can be derived from existing rules of the system. It can be an existing rule with more stringent predicate or a sequential combination of several existing rules. For example, the following rule allows a sache, in one rewriting step, to write the data of a dirty copy back to the memory and purge the cell from the sache. It can be derived by applying the writeback rule and the purge rule consecutively.

CRF-Flush Rule

$$\begin{array}{ll} & \text{Sys}(\mathit{mem}, \text{ Site}(\text{Cell}(a,v,\text{Dirty}) \mid \mathit{sache}, \ \mathit{pmb}, \ \mathit{mpb}, \ \mathit{proc}) \mid \mathit{sites}) \\ \rightarrow & \text{Sys}(\mathit{mem}[a{:=}v], \text{ Site}(\mathit{sache}, \ \mathit{pmb}, \ \mathit{mpb}, \ \mathit{proc}) \mid \mathit{sites}) \end{array}$$

3.2.1 Stalled Instruction Rules

The specification of CRF includes only a set of imperative rules; this intentionally does not address certain implementation issues. For example, when a processor intends to execute a Commit instruction on a dirty cell, it is stalled until the dirty copy is written back to the memory. This requires that the writeback rule be invoked in order to complete the stalled instruction. In general, appropriate background operations must be invoked when an instruction is stalled so that the processor can make progress eventually. The stalled instruction rules serve such a purpose.

CRF-Loadl-on-Invalid Rule

$$\begin{array}{ll} & \text{Sys}(\mathit{mem}, \text{ Site}(\mathit{sache}, \ \langle t,\text{Loadl}(a)\rangle;\mathit{pmb}, \ \mathit{mpb}, \ \mathit{proc}) \mid \mathit{sites}) \qquad \mathit{if} \quad a \notin \mathit{sache} \\ \rightarrow & \text{Sys}(\mathit{mem}, \text{ Site}(\text{Cell}(a,\mathit{mem}[a],\text{Clean}) \mid \mathit{sache}, \ \mathit{pmb}, \ \mathit{mpb} | \langle t,\mathit{mem}[a]\rangle, \ \mathit{proc}) \mid \mathit{sites}) \end{array}$$

CRF-Storel-on-Invalid Rule

$\quad\text{Site}(sache,\ \langle t,\text{Storel}(a,v)\rangle;pmb,\ mpb,\ proc) \qquad \textit{if} \quad a \notin sache$

$\rightarrow \quad \text{Site}(\text{Cell}(a,v,\text{Dirty}) \mid sache,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc)$

CRF-Commit-on-Dirty Rule

$\quad\text{Sys}(mem,\ \text{Site}(\text{Cell}(a,v,\text{Dirty}) \mid sache,\ \langle t,\text{Commit}(a)\rangle;pmb,\ mpb,\ proc) \mid sites)$

$\rightarrow \quad \text{Sys}(mem[a{:=}v],\ \text{Site}(\text{Cell}(a,v,\text{Clean}) \mid sache,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc) \mid sites)$

CRF-Reconcile-on-Clean Rule

$\quad\text{Site}(\text{Cell}(a,\text{-},\text{Clean}) \mid sache,\ \langle t,\text{Reconcile}(a)\rangle;pmb,\ mpb,\ proc)$

$\rightarrow \quad \text{Site}(sache,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc)$

It is obvious that the stalled instruction rules given above can be derived from CRF rules:

CRF-Loadl-on-Invalid = *CRF-Cache* + *CRF-Loadl*
CRF-Storel-on-Invalid = *CRF-Cache* + *CRF-Storel*
CRF-Commit-on-Dirty = *CRF-Writeback* + *CRF-Commit*
CRF-Reconcile-on-Clean = *CRF-Purge* + *CRF-Reconcile*

3.2.2 Relaxed Execution Rules

CRF gives precise conditions under which memory instructions can be reordered. In CRF, an instruction must be brought to the front of the processor-to-memory buffer before it can be executed. This constraint can be relaxed without affecting program behaviors. For example, a Loadl instruction can be performed if there is no preceding Storel or Reconcile instruction to the same address. It is easy to derive the predicates under which a CRF instruction can be performed in the presence of other outstanding instructions:

- Loadl(a): no preceding Storel(a,-) or Reconcile(a)
- Storel(a,-): no preceding Loadl(a), Storel(a,-) or Fence$_{\alpha\mathrm{w}}$(-,a)
- Commit(a): no preceding Storel(a,-)
- Reconcile(a): no preceding Fence$_{\alpha\mathrm{r}}$(-,a)
- Fence$_{\mathrm{r}\alpha}$(a,-): no preceding Loadl(a)
- Fence$_{\mathrm{w}\alpha}$(a,-): no preceding Commit(a)

The relaxed execution rules below allow a memory instruction to be performed before its preceding instructions are completed:

CRF-Relaxed-Loadl Rule

$\quad\text{Site}(sache,\ pmb_1;\langle t,\text{Loadl}(a)\rangle;pmb_2,\ mpb,\ proc)$

$\qquad\qquad \textit{if} \quad \text{Cell}(a,v,\text{-}) \in sache \ \wedge \ \text{Storel}(a,\text{-}),\ \text{Reconcile}(a) \notin pmb_1$

$\rightarrow \quad \text{Site}(sache,\ pmb_1;pmb_2,\ mpb|\langle t,v\rangle,\ proc)$

CRF-Relaxed-Storel Rule

$\quad\text{Site}(\text{Cell}(a,\text{-},\text{-}) \mid sache,\ pmb_1;\langle t,\text{Storel}(a,v)\rangle;pmb_2,\ mpb,\ proc)$

$\qquad\qquad \textit{if} \quad \text{Loadl}(a),\ \text{Storel}(a,\text{-}),\ \text{Fence}_{\alpha\mathrm{w}}(\text{-},a) \notin pmb_1$

$\rightarrow \quad \text{Site}(\text{Cell}(a,v,\text{Dirty}) \mid sache,\ pmb_1;pmb_2,\ mpb|\langle t,\text{Ack}\rangle,\ proc)$

CRF-Relaxed-Commit Rule

$\text{Site}(\mathit{sache},\ \mathit{pmb}_1;\langle t,\text{Commit}(a)\rangle;\mathit{pmb}_2,\ \mathit{mpb},\ \mathit{proc})$

$\mathit{if}\quad \text{Cell}(a,\text{-},\text{Dirty}) \notin \mathit{sache} \ \wedge\ \text{Store}(a,\text{-}) \notin \mathit{pmb}_1$

$\rightarrow\quad \text{Site}(\mathit{sache},\ \mathit{pmb}_1;\mathit{pmb}_2,\ \mathit{mpb}|\langle t,\text{Ack}\rangle,\ \mathit{proc})$

CRF-Relaxed-Reconcile Rule

$\text{Site}(\mathit{sache},\ \mathit{pmb}_1;\langle t,\text{Reconcile}(a)\rangle;\mathit{pmb}_2,\ \mathit{mpb},\ \mathit{proc})$

$\mathit{if}\quad \text{Cell}(a,\text{-},\text{Clean}) \notin \mathit{sache} \ \wedge\ \text{Fence}_{\alpha\text{r}}(\text{-},a) \notin \mathit{pmb}_1$

$\rightarrow\quad \text{Site}(\mathit{sache},\ \mathit{pmb}_1;\mathit{pmb}_2,\ \mathit{mpb}|\langle t,\text{Ack}\rangle,\ \mathit{proc})$

CRF-Relaxed-Fence$_{r\alpha}$ Rule

$\text{Site}(\mathit{sache},\ \mathit{pmb}_1;\langle t,\text{Fence}_{\text{r}\alpha}(a,\text{-})\rangle;\mathit{pmb}_2,\ \mathit{mpb},\ \mathit{proc})\qquad \mathit{if}\quad \text{Loadl}(a) \notin \mathit{pmb}_1$

$\rightarrow\quad \text{Site}(\mathit{sache},\ \mathit{pmb}_1;\mathit{pmb}_2,\ \mathit{mpb}|\langle t,\text{Ack}\rangle,\ \mathit{proc})$

CRF-Relaxed-Fence$_{w\alpha}$ Rule

$\text{Site}(\mathit{sache},\ \mathit{pmb}_1;\langle t,\text{Fence}_{\text{w}\alpha}(a,\text{-})\rangle;\mathit{pmb}_2,\ \mathit{mpb},\ \mathit{proc})\qquad \mathit{if}\quad \text{Commit}(a) \notin \mathit{pmb}_1$

$\rightarrow\quad \text{Site}(\mathit{sache},\ \mathit{pmb}_1;\mathit{pmb}_2,\ \mathit{mpb}|\langle t,\text{Ack}\rangle,\ \mathit{proc})$

The relaxed execution rules imply that the processor can dispatch CRF instructions out-of-order without affecting program behaviors in multiprocessor systems. For example, a Loadl instruction can be dispatched as long as there is no preceding Storel or Reconcile instruction to the same address which has not been dispatched.

3.3 Coarse-grain CRF Instructions

In CRF, it is always safe to insert Commit, Reconcile and Fence instructions without affecting the correctness of the program. This is because a dirty copy can always be written back to the memory, a clean copy can always be purged, and a sequential execution of instructions always produces a correct result. An extra Commit, Reconcile or Fence instruction may eliminate a legal program behavior but can never admit an illegal program behavior. This fact allows us to introduce coarse-grain versions of such instructions, which are more practical at the ISA level.

A coarse-grain fence imposes an ordering constraint with respect to address ranges, instead of individual locations. For example, $\text{Fence}_{\text{rw}}(A_1,A_2)$ ensures that all preceding Loadl operations to address range A_1 must be performed before any following Storel operation to address range A_2 can be performed. It can be defined in terms of $|A_1|\cdot|A_2|$ fine-grain fences that obey all the reordering rules given earlier. Similarly, $\text{Commit}(A)$ and $\text{Reconcile}(A)$ can be defined in terms of fine-grain commit and reconcile operations, respectively. An address range can be a cache line, a page or even the whole address space (we represent the address space by '$*$').

Of particular interest are memory fences that impose ordering constraints between some memory range and an individual location. For example, we can define the following coarse-grain fences:

$$\begin{array}{lcl}
\text{PreFenceR}(a) & \equiv & \text{Fence}_{\text{rr}}(*,a);\ \text{Fence}_{\text{wr}}(*,a) \\
\text{PreFenceW}(a) & \equiv & \text{Fence}_{\text{rw}}(*,a);\ \text{Fence}_{\text{ww}}(*,a) \\
\text{PostFenceR}(a) & \equiv & \text{Fence}_{\text{rr}}(a,*);\ \text{Fence}_{\text{rw}}(a,*) \\
\text{PostFenceW}(a) & \equiv & \text{Fence}_{\text{wr}}(a,*);\ \text{Fence}_{\text{ww}}(a,*)
\end{array}$$

Informally, PreFenceW(a) requires that all preceding Loadl and Commit instructions be completed before any following Storel instruction to location a can be performed. PostFenceR(a) requires that all preceding Loadl instructions to location a be completed before any following Reconcile or Storel instruction can be performed. In an implementation of release consistency, we can use a PreFenceW instruction before releasing a semaphore, and a PostFenceR instruction after acquiring a semaphore.

CRF-bits The Loadl and Storel instructions can be augmented with proper CRF-bits so that the semantics of some commit, reconcile and fence operations can be represented without explicit instructions. There are six CRF-bits, Com, Rec, PreR, PreW, PostR and PostW. If turned on, the Com-bit implies a commit operation for a Storel, while a Rec-bit implies a reconcile operation for a Loadl.

The PreR, PreW, PostR and PostW bits can be used to enforce fence operations between an address and the whole address space. Intuitively, the PreR-bit means that all preceding Loadl instructions must complete before the instruction itself completes; while the PreW-bit means that all preceding Commit instructions must complete before the instruction itself completes. For a Loadl instruction, it makes little sense to set the PreR-bit or PreW-bit without setting the Rec-bit. The PostR-bit implies that the instruction must complete before any following Reconcile instruction completes; while the PostW-bit implies that the instruction must complete before any following Storel instruction completes. For a Storel instruction, it makes little sense to set the PostR-bit or PostW-bit without setting the Com-bit. The precise semantics of the CRF-bits can be given using CRF instructions. For example, an instruction with all the CRF-bits set can be defined as follows:

$$\begin{array}{l}
\text{Loadl}(a)\ [\text{Rec,PreR,PreW,PostR,PostW}] \\
\quad \equiv\ \text{Fence}_{\text{rr}}(*,a);\ \text{Fence}_{\text{wr}}(*,a);\ \text{Reconcile}(a);\ \text{Loadl}(a);\ \text{Fence}_{\text{rr}}(a,*);\ \text{Fence}_{\text{rw}}(a,*) \\
\text{Storel}(a,v)\ [\text{Com,PreR,PreW,PostR,PostW}] \\
\quad \equiv\ \text{Fence}_{\text{rw}}(*,a);\ \text{Fence}_{\text{ww}}(*,a);\ \text{Storel}(a,v);\ \text{Commit}(a);\ \text{Fence}_{\text{wr}}(a,*);\ \text{Fence}_{\text{ww}}(a,*)
\end{array}$$

While fine-grain commit and reconcile instructions give the compiler more control over coherence actions of the system, coarse-grain fence instructions can reduce the number of instructions to be executed. In practice, commit and reconcile instructions can also be merged with fence instructions under certain circumstances. Furthermore, the semantics of some CRF instructions can be attached to synchronization instructions such as Test-&-Set and Swap.

3.4 Universality of the CRF Model

Most relaxed or weaker memory models have arisen as a consequence of specific architectural optimizations in the implementation of memory access instructions, rather than from some high-level design. Different manufacturers have different memory models; even the same manufacturer can have different memory models for different generations of machines. The CRF model can be used to eliminate the *modèle de l'année* aspect of many existing memory models. Programs written under sequential consistency and various weaker memory models can be translated into CRF programs without incurring unnecessary overhead. The translations can be taken as precise definitions of the often imprecise descriptions of the semantics of memory instructions given by the manufacturers. On the other hand, CRF programs can be mapped back to programs that can run efficiently on existing microprocessors. This section demonstrates the upward and downward compatibility of the CRF model.

The upward compatibility refers to the the ability to run existing programs correctly and efficiently on a CRF machine. As an example, we show translation schemes for SC, Sparc's TSO, PSO and RMO models and the IBM 370's model. Weaker memory models that allow memory instructions to be reordered usually provide memory barrier or fence instructions that can be used to enforce necessary ordering. Some machines such as IBM 370 have no explicit barrier instructions but instead rely on the implicit barrier-like semantics of some special instructions. We still refer to such instructions as memory barriers (Membar) in the translation schemes.

- The translation from SC to CRF is simple because SC requires strict in-order and atomic execution of load and store instructions.

  $$\begin{aligned}
  &\text{Load}_{\text{sc}}(a) &&\equiv\quad \text{Fence}_{\text{rr}}(*,a);\ \text{Fence}_{\text{wr}}(*,a);\ \text{Reconcile}(a);\ \text{Loadl}(a)\\
  &\text{Store}_{\text{sc}}(a,v) &&\equiv\quad \text{Fence}_{\text{rw}}(*,a);\ \text{Fence}_{\text{ww}}(*,a);\ \text{Storel}(a,v);\ \text{Commit}(a)
  \end{aligned}$$

  The translation guarantees that the resulting CRF program has exactly the same behavior as the SC program. Note that each fence can be replaced by a coarse-grain instruction Fence($*$,$*$) without eliminating any program behavior.

- TSO allows a load to be performed before outstanding stores complete, which virtually models FIFO write-buffers. It also allows a load to retrieve the data from an outstanding store to the same address before the data is observable to other processors.

  $$\begin{aligned}
  &\text{Load}_{\text{tso}}(a) &&\equiv\quad \text{Fence}_{\text{rr}}(*,a);\ \text{Reconcile}(a);\ \text{Loadl}(a)\\
  &\text{Store}_{\text{tso}}(a,v) &&\equiv\quad \text{Fence}_{\text{rw}}(*,a);\ \text{Fence}_{\text{ww}}(*,a);\ \text{Storel}(a,v);\ \text{Commit}(a)\\
  &\text{Membar-like}_{\text{tso}} &&\equiv\quad \text{Fence}_{\text{wr}}(*,*)
  \end{aligned}$$

  The translation places a Fence_{rr} before each Reconcile-Loadl pair to ensure the load-load ordering, and a Fence_{rw} and a Fence_{ww} before each Storel-Commit pair to ensure the load-store and store-store ordering. The Membar instruction is simply translated into a write-read fence for all the addresses.

- PSO allows a load to overtake outstanding stores and in addition, a store to overtake other outstanding stores. This models non-FIFO write buffers. The short-circuiting is still permitted as in TSO.

$$\begin{aligned}
&\text{Load}_{\text{pso}}(a) &&\equiv\quad \text{Fence}_{\text{rr}}(*,a);\ \text{Reconcile}(a);\ \text{Loadl}(a)\\
&\text{Store}_{\text{pso}}(a,v) &&\equiv\quad \text{Fence}_{\text{rw}}(*,a);\ \text{Storel}(a,v);\ \text{Commit}(a)\\
&\text{Membar}_{\text{pso}} &&\equiv\quad \text{Fence}_{\text{wr}}(*,*);\ \text{Fence}_{\text{ww}}(*,*)
\end{aligned}$$

- RMO allows memory accesses to be reordered arbitrarily, provided that data dependencies are respected.

$$\begin{aligned}
&\text{Load}_{\text{rmo}}(a) &&\equiv\quad \text{Reconcile}(a);\ \text{Loadl}(a)\\
&\text{Store}_{\text{rmo}}(a,v) &&\equiv\quad \text{Storel}(a,v);\ \text{Commit}(a)\\
&\text{Membar\#LoadLoad}_{\text{rmo}} &&\equiv\quad \text{Fence}_{\text{rr}}(*,*)\\
&\text{Membar\#LoadStore}_{\text{rmo}} &&\equiv\quad \text{Fence}_{\text{rw}}(*,*)\\
&\text{Membar\#StoreLoad}_{\text{rmo}} &&\equiv\quad \text{Fence}_{\text{wr}}(*,*)\\
&\text{Membar\#StoreStore}_{\text{rmo}} &&\equiv\quad \text{Fence}_{\text{ww}}(*,*)
\end{aligned}$$

- Like TSO, IBM 370 allows the use of write buffers so that a load can be performed before outstanding stores complete. However, it prohibits data short-circuiting from write buffers and requires that each load retrieve data directly from the memory. In other words, the data of a store operation cannot be observed by any processor before it becomes observable to all the processors. The translation from IBM 370 to CRF is the same as the translation from TSO to CRF, except for a Fence$_{\text{wr}}$ instruction to ensure the ordering between a Store and a following Load to the same address.

$$\begin{aligned}
&\text{Load}_{370}(a) &&\equiv\quad \text{Fence}_{\text{wr}}(a,a);\ \text{Fence}_{\text{rr}}(*,a);\ \text{Reconcile}(a);\ \text{Loadl}(a)\\
&\text{Store}_{370}(a,v) &&\equiv\quad \text{Fence}_{\text{rw}}(*,a);\ \text{Fence}_{\text{ww}}(*,a);\ \text{Storel}(a,v);\ \text{Commit}(a)\\
&\text{Membar}_{370} &&\equiv\quad \text{Fence}_{\text{wr}}(*,*)
\end{aligned}$$

Downward compatibility refers to the ability to run CRF programs on existing machines. Many existing systems can be interpreted as specific implementations of CRF, though more efficient implementations are possible. We show how CRF programs can be translated into programs that can run on microprocessors that support SC, Sparc's TSO, PSO and RMO models, and IBM 370's model. In all the translation schemes, we translate Loadl and Storel into the load and store instructions of the target machines. Both Commit and Reconcile become Nop's, since the semantics are implied by the corresponding load and store operations. For different target machines, the translation schemes differ in dealing with memory fences. A translation avoids employing unnecessary memory barriers by exploiting specific ordering constraints guaranteed by the underlying architecture.

- Translation from CRF to SC: All CRF fences simply become Nop's, since memory accesses are executed strictly in-order in SC machines.

- Translation from CRF to TSO: The Fence$_{\text{rr}}$, Fence$_{\text{rw}}$ and Fence$_{\text{ww}}$ instructions are translated into Nop's, since TSO implicitly guarantees the load-load, load-store and store-store ordering. The Fence$_{\text{wr}}$ instruction is translated into Membar.

- Translation from CRF to PSO: The Fence_{rr} and Fence_{rw} instructions are translated into Nop's, since PSO implicitly preserves the load-load and load-store ordering. Both Fence_{wr} and Fence_{ww} are translated into Membar.
- Translation from CRF to RMO: Since RMO assumes no ordering between memory accesses unless explicit memory barriers are inserted, the translation scheme translates each CRF fence into the corresponding RMO memory barrier.
- Translation from CRF to IBM 370: Since IBM 370's model is slightly stricter than the TSO model, the translation can avoid some memory barriers by taking advantage from the fact that IBM 370 prohibits data short-circuiting from write-buffers. This suggests that a Fence_{wr} with identical pre-address and post-address be translated into a Nop.

We can translate programs based on release consistency to CRF programs by defining the release and acquire operations using CRF instructions and synchronization instructions such as Lock/Unlock. The synchronization instructions are provided for mutual exclusion, and have no ordering implication on other instructions. In terms of reordering constraints, a Lock can be treated as a Loadl followed by a Storel, and an Unlock can be treated as a Storel.

$$\begin{aligned} \text{Release(s)} &\equiv \quad \text{Commit}(*);\ \text{PreFenceW(s)};\ \text{Unlock(s)} \\ \text{Acquire(s)} &\equiv \quad \text{Lock(s)};\ \text{PostFenceR(s)};\ \text{Reconcile}(*) \end{aligned}$$

One can think of the translation scheme above as the definition of one version of release consistency. Obviously, memory accesses after a release can be performed before the semaphore is released, because the release only imposes a pre-fence on preceding accesses. Similarly, memory accesses before an acquire do not have to be completed before the semaphore is acquired, because the acquire only imposes a post-fence on following memory accesses. Furthermore, modified data of a store before a release need to be written back to the memory at the release point, but a data copy of the address in another cache does not have to be invalidated or updated immediately. This is because the stale data copy, if any, will be reconciled at the next acquire point.

3.5 The Generalized CRF Model

In CRF, the memory behaves as the rendezvous of saches. A writeback operation is atomic with respect to all the sites: if a sache can retrieve some data from the memory, another sache must be able to retrieve the same data from the memory at the same time. Therefore, if two stores are observed by more than one processor, they are observed in the same order provided that the loads used in the observation are executed in order.

As an example, the following program illustrates that non-atomic store operations can generate surprising behaviors even when only one memory location is involved. Assume initially location a has value 0. Processors A and B modify location a while processors C and D read it. An interesting question is whether it is possible that processor C obtains 1 and 2, while processor D obtains 2 and 1. This cannot happen in CRF.

Processor A	*Processor B*	*Processor C*	*Processor D*
Storel(a,1);	Storel(a,2);	Reconcile(a);	Reconcile(a);
Commit(a);	Commit(a);	r_1 = Loadl(a);	r_2 = Loadl(a);
		Fence$_{\text{rr}}$(a,a);	Fence$_{\text{rr}}$(a,a);
		Reconcile(a);	Reconcile(a);
		r_3 = Loadl(a);	r_4 = Loadl(a);

The atomicity of writeback operations is a deliberate design choice to avoid unnecessary semantic complications without compromising implementation flexibility. Enforcing such atomicity is easy in directory-based DSM systems where the home site can behave as the rendezvous. However, such semantics may prohibit CRF from representing certain memory models that allow non-atomic stores. Furthermore, a memory model with non-atomic stores may allow more implementation flexibility for cache coherence protocols.

The GCRF (Generalized Commit-Reconcile & Fences) model allows writeback operations to be performed in some non-atomic fashion. Figure 3.7 gives the system configuration of GCRF. The system contains a set of sites, where each site has a site identifier, a memory and a semantic cache (sache). Unlike CRF in which a memory is shared by all the sites, GCRF maintains a memory at in each site. A sache can can retrieve data from its own memory, but can write a dirty copy back to any memory. The writeback operations with respect to different memories can be performed non-atomically so that writeback operations for different addresses can be interleaved with each other.

For each sache cell, the sache state is a directory that records the site identifiers whose memories need to be updated. If the directory is empty, it means the data has not been modified since it was cached or the modified data has been written back to all the memories. When a sache obtains a data copy from its memory, it sets the directory to empty. When a sache cell is modified, the directory records the set of all the site identifiers. When a sache writes a dirty copy back to a memory, it removes the corresponding identifier from the directory to ensure that the same data can be written back to each memory at most once.

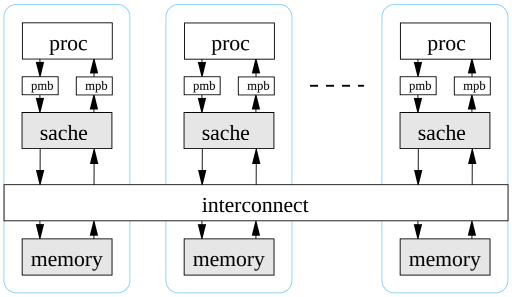
Loadl and Storel Rules: A Loadl or Storel can be performed if the address is cached in the sache. Once a Storel is performed, the sache state records all site identifiers since the dirty data has not been written back to any memory (notation 'S' represents the set of identifiers for all the sites).

GCRF-Loadl Rule

$$\begin{array}{ll} & \text{Site}(id,\ mem,\ sache,\ \langle t,\text{Loadl}(a)\rangle;pmb,\ mpb,\ proc) \qquad \textit{if} \quad \text{Cell}(a,v,\text{-}) \in sache \\ \rightarrow & \text{Site}(id,\ mem,\ sache,\ pmb,\ mpb|\langle t,v\rangle,\ proc) \end{array}$$

GCRF-Storel Rule

$$\begin{array}{ll} & \text{Site}(id,\ mem,\ \text{Cell}(a,\text{-},\text{-})\ |\ sache,\ \langle t,\text{Storel}(a,v)\rangle;pmb,\ mpb,\ proc) \\ \rightarrow & \text{Site}(id,\ mem,\ \text{Cell}(a,v,\text{S})\ |\ sache,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc) \end{array}$$



SYS	$\equiv$	SITEs	*System*
SITEs	$\equiv$	SITE $[\!]$ SITE $\mid$ SITEs	*Set of Sites*
SITE	$\equiv$	Site(id, MEM, SACHE, PMB, MPB, PROC)	*Site*
SACHE	$\equiv$	ϵ $[\!]$ Cell(a,v,CSTATE) $\mid$ SACHE	*Semantic Cache*
CSTATE	$\equiv$	ϵ $[\!]$ $id$$\mid$CSTATE	*Cache State*

Figure 3.7: System Configuration of GCRF

Commit and Reconcile Rules: A Commit can be completed if the address is uncached or the data has been written back to all the sites since its most recent update. A Reconcile can be completed if the address is uncached or the modified data has not been written back to all the sites.

GCRF-Commit Rule

$$\begin{array}{ll} & \text{Site}(id,\ mem,\ sache,\ \langle t,\text{Commit}(a)\rangle;pmb,\ mpb,\ proc) \qquad \textit{if} \quad a \notin sache \ \vee\ \text{Cell}(a,\text{-},\epsilon) \in sache \\ \rightarrow & \text{Site}(id,\ mem,\ sache,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc) \end{array}$$

GCRF-Reconcile Rule

$$\begin{array}{ll} & \text{Site}(id,\ mem,\ sache,\ \langle t,\text{Reconcile}(a)\rangle;pmb,\ mpb,\ proc) \qquad \textit{if} \quad \text{Cell}(a,\text{-},\epsilon) \notin sache \\ \rightarrow & \text{Site}(id,\ mem,\ sache,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc) \end{array}$$

Cache, Writeback and Purge Rules: A sache can obtain a clean copy from its own memory if the address is not cached at the time. A sache can write the data of a cell back to a memory that has not been updated according to the sache state. A sache can purge a cell if the data has been written back to all the sites since its most recent update.

GCRF-Cache Rule

$$\begin{array}{ll} & \text{Site}(id,\ mem,\ sache,\ pmb,\ mpb,\ proc) \qquad \textit{if} \quad a \notin sache \\ \rightarrow & \text{Site}(id,\ mem,\ \text{Cell}(a,mem[a],\epsilon) \mid sache,\ pmb,\ mpb,\ proc) \end{array}$$

Processor Rules				
Rule Name	**Instruction**	**Cstate**	**Action**	**Next Cstate**
GCRF-Loadl	Loadl(a)	Cell(a,v,cs)	*retire*	Cell(a,v,cs)
GCRF-Storel	Storel(a,v)	Cell(a,-,-)	*retire*	Cell(a,v,S)
GCRF-Commit	Commit(a)	Cell(a,v,ϵ)	*retire*	Cell(a,v,ϵ)
		$a \notin sache$	*retire*	$a \notin sache$
GCRF-Reconcile	Reconcile(a)	Cell(a,v,cs) ($cs \neq \epsilon$)	*retire*	Cell(a,v,cs)
		$a \notin sache$	*retire*	$a \notin sache$

Background Rules				
Rule Name	**Cstate**	**Mstate**	**Next Cstate**	**Next Mstate**
GCRF-Cache	$a \notin sache$	Cell(a,v)	Cell(a,v,ϵ)	Cell(a,v)
GCRF-Writeback	Cell(a,v,id|cs)	Cell(a,-)	Cell(a,v,cs)	Cell(a,v)
GCRF-Purge	Cell(a,-,ϵ)	Cell(a,v)	$a \notin sache$	Cell(a,v)

Figure 3.8: Summary of GCRF Rules (except reordering rules)

GCRF-Writeback Rule

$$
\begin{array}{ll}
 & \text{Site}(id,\ mem,\ \text{Cell}(a,v,id|cs)\,|\,sache,\ pmb,\ mpb,\ proc) \\
\rightarrow & \text{Site}(id,\ mem[a{:=}v],\ \text{Cell}(a,v,cs)\,|\,sache,\ pmb,\ mpb,\ proc)
\end{array}
$$

$$
\begin{array}{ll}
 & \text{Site}(id,\ mem,\ \text{Cell}(a,v,id_1|cs)\,|\,sache,\ pmb,\ mpb,\ proc)\ \ | \\
 & \qquad\qquad \text{Site}(id_1,\ mem_1,\ sache_1,\ pmb_1,\ mpb_1,\ proc_1) \\
\rightarrow & \text{Site}(id,\ mem,\ \text{Cell}(a,v,cs)\,|\,sache,\ pmb,\ mpb,\ proc)\ \ | \\
 & \qquad\qquad \text{Site}(id_1,\ mem_1[a{:=}v],\ sache_1,\ pmb_1,\ mpb_1,\ proc_1)
\end{array}
$$

GCRF-Purge Rule

$$
\begin{array}{ll}
 & \text{Site}(id,\ mem,\ \text{Cell}(a,\text{-},\epsilon)\,|\,sache,\ pmb,\ mpb,\ proc) \\
\rightarrow & \text{Site}(id,\ mem,\ sache,\ pmb,\ mpb,\ proc)
\end{array}
$$

The GCRF model also allows instructions to be reordered; the reordering rules of CRF all remain unchanged. The GCRF rules are summarized in Figure 3.8. In the tabular description, for the cache and purge rules, the memory state reflects the memory cell in the same site as the sache cell; for the writeback rule, the memory state reflects the memory cell in site id.

The GCRF model also allows instructions to be reordered; the reordering rules are exactly the same as those of CRF. In addition, we can define a new commit instruction that commits an address with respect to an individual memory. The semantics of the instruction can be specified as follows:

$$
\begin{array}{ll}
 & \text{Site}(id,\ mem,\ sache,\ \langle t,\text{Commit}(a,id)\rangle;pmb,\ mpb,\ proc) \qquad \textit{if} \quad \text{Cell}(a,\text{-},id|\text{-}) \notin sache \\
\rightarrow & \text{Site}(id,\ mem,\ sache,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc)
\end{array}
$$

The Commit(a,id) instruction guarantees that the dirty copy is written back to the memory at site id before the instruction completes. The normal commit instruction can be defined by a sequence of the finer-grain commit instructions.

Chapter 4

The Base Cache Coherence Protocol

The Base protocol is the most straightforward implementation of the CRF model. An attractive characteristic of Base is its simplicity: no state needs to be maintained at the memory side. In Base, a commit operation on a dirty cell forces the data to be written back to the memory, and a reconcile operation on a clean cell forces the data to be purged from the cache. This is ideal for programs in which only necessary commit and reconcile operations are performed.

In this chapter, we first present a novel protocol design methodology called Imperative-&-Directive that will be used throughout this thesis. Section 4.2 describes the system configuration of the Base protocol and gives the message passing rules for non-FIFO and FIFO networks. We define the imperative rules of Base in Section 4.3, and present the complete Base protocol in Section 4.4. Section 4.5 proves the soundness of Base by showing that the imperative Base rules can be simulated in CRF; Section 4.6 proves the liveness of Base by showing that each processor can always make progress (that is, a memory instruction can always be completed eventually).

4.1 The Imperative-&-Directive Design Methodology

In spite of the development of various cache coherence protocols, it is difficult to discern any methodology that has guided the design of existing protocols. A major source of complexity in protocol design is that the designer often deals with many different issues simultaneously. Are coherence states being maintained correctly? Is it possible that a cache miss may never be serviced? What is the consequence if messages get reordered in the network? How to achieve better adaptivity for programs with different access patterns? Answering such questions can be difficult for sophisticated protocols with various optimizations. The net result is that protocol design is viewed as an enigma, and even the designer is not totally confident of his or her understanding of the protocol behavior.

We propose a two-stage design methodology called Imperative-&-Directive that separates soundness and liveness concerns in the design process (see Figure 4.1). Soundness ensures that

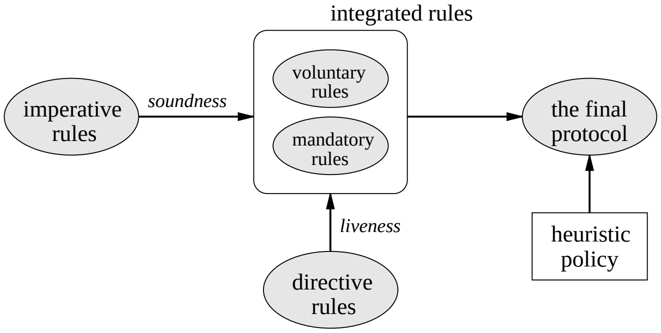


Figure 4.1: The Imperative-&-Directive Design Methodology

the system only exhibits legal behaviors permitted by the specification, and liveness ensures that the system takes desirable actions eventually to make progress. The first stage of the design involves only *imperative rules* that specify coherence actions that can affect the soundness of the system. The second stage of the design involves *directive rules* that can be used to invoke imperative rules, and the integration of imperative and directive rules to ensure both the soundness and liveness of the protocol. The rules of the integrated protocol can be derived from the imperative and directive rules.

Protocol messages can be classified as imperative messages and directive messages accordingly. Imperative rules use imperative messages to propagate data or other information that determine the soundness of the system, and directive rules use directive messages to invoke imperative actions at remote sites. In the integrated protocol, directive messages behave as extra predicates on imperative actions. Since directive rules are prohibited from modifying soundness states, improper conditions for invoking imperative rules can cause deadlock or livelock but cannot affect soundness. Therefore, it suffices to verify the soundness of the system with respect to the imperative rules, rather than the integrated rules of the integrated protocol.

As an example, consider an imperative rule that allows a cache to write a dirty copy back to the memory via an imperative writeback message. The imperative rule does not specify under what conditions it must be invoked to ensure the liveness of the system. When the memory wants the cache to perform a writeback operation, it sends a directive writeback request to the cache. The integrated protocol ensures both soundness and liveness by requiring that the cache write the data copy back to the memory once a writeback request is received.

The Imperative-&-Directive methodology can dramatically simplify the design and verification of cache coherence protocols. Protocols designed with this methodology are often easy to understand and modify. The methodology will be applied to the design of all the protocols presented in this thesis.

Mandatory Rules and Voluntary Rules To ensure liveness, we need to guarantee some fairness properties for the integrated rules in the integrated protocol. The integrated rules can be classified into two non-overlapping sets: *mandatory rules* and *voluntary rules*. The distinction between mandatory rules and voluntary rules is that mandatory rules require at least weak fairness to ensure the liveness of the system, while voluntary rules have no such requirement and are provided purely for adaptivity and performance reason. Intuitively, an enabled mandatory rule must be applied eventually, while an enabled voluntary rule may or may not be applied eventually.

The fairness requirement of mandatory rules can be expressed in terms of weak fairness and strong fairness. Weak fairness means that if a mandatory rule can be applied, it will be applied eventually or become impossible to apply at some later time. Strong fairness means that if a mandatory rule can be applied, it will be applied eventually or become impossible to apply forever. When we say a rule is weakly or strongly fair, we mean the application of the rule on each redex is weakly or strongly fair.

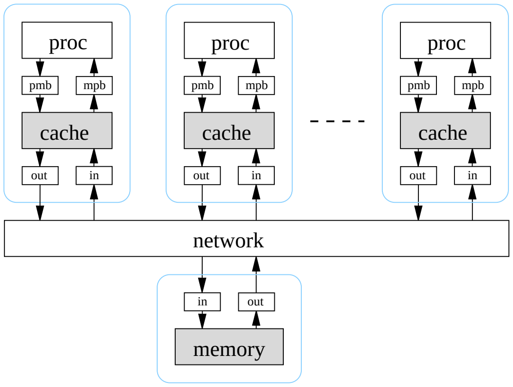
A mandatory action is usually enabled by events such as an instruction from the processor or a message from the network. A voluntary action, in contrast, can be enabled as long as the cache or memory cell is in some appropriate state. For example, a mandatory writeback rule requires a cache to write a dirty copy back to the memory once a writeback request is received, while a voluntary writeback rule allows the same operation as long as the cache state of the address shows that the data has been modified.

Conventional cache coherence protocols consist of only mandatory actions. Our view of an adaptive coherence protocol consists of three components, mandatory rules, voluntary rules and heuristic policies. The existence of voluntary rules provides enormous adaptivity that can be exploited via various heuristic policies. A heuristic mechanism can use heuristic messages and heuristic states to help determine when a voluntary rule should be invoked. Different heuristic policies can result in different performance, but the soundness and liveness of the system are always guaranteed.

4.2 The Message Passing Rules

Figure 4.2 defines the system configuration of Base. The system contains a memory site and a set of cache sites. The memory site has three components, a memory, an incoming queue and an outgoing queue. Each cache site contains an identifier, a cache, an incoming queue, an outgoing queue and a processor. Although the memory site appears as one component syntactically, it can be distributed among multiple sites in DSM systems. Initially all caches and message queues are empty.

A message queue is a sequence of messages. We use '$\odot$' and '$\otimes$' as the constructor for incoming and outgoing queues, respectively. Each message has several fields: a source, a destination, a command, an address and an optional data. The source and destination can be either a cache site or the home memory (represented by 'H'). Given a message *msg*, we use notations Src(*msg*), Dest(*msg*), Cmd(*msg*) and Addr(*msg*) to represent its source, destination, command and address.



SYS	$\equiv$	Sys(MSITE, SITEs)	*System*
MSITE	$\equiv$	Msite(MEM, IN, OUT)	*Memory Site*
MEM	$\equiv$	ϵ $\parallel$ Cell(a,v) | MEM	*Memory*
SITEs	$\equiv$	SITE $\parallel$ SITE | SITEs	*Set of Cache Sites*
SITE	$\equiv$	Site(id, CACHE, IN, OUT, PMB, MPB, PROC)	*Cache Site*
CACHE	$\equiv$	ϵ $\parallel$ Cell(a,v,CSTATE) | CACHE	*Cache*
IN	$\equiv$	ϵ $\parallel$ MSG $\odot$ IN	*Incoming Queue*
OUT	$\equiv$	ϵ $\parallel$ MSG $\otimes$ OUT	*Outgoing Queue*
MSG	$\equiv$	Msg(src,$dest$,CMD,a,v)	*Message*
CSTATE	$\equiv$	Clean $\parallel$ Dirty $\parallel$ CachePending $\parallel$ WbPending	*Cache State*
CMD	$\equiv$	CacheReq $\parallel$ Cache $\parallel$ Wb $\parallel$ WbAck	*Command*

Figure 4.2: System Configuration of Base

Message passing can happen between a cache and the memory. A message passing rule delivers a message from the source's outgoing queue to the destination's incoming queue. Message passing can be non-FIFO or FIFO. Non-FIFO message passing allows messages to be delivered in arbitrary order, while FIFO message passing ensures messages between the same source and destination to be received in the order in which they are issued.

Message-Cache-to-Mem Rule

$\quad$ Sys(Msite(mem, min, $mout$), Site(id, $cache$, cin, $msg \otimes cout$, pmb, mpb, $proc$) | $sites$)

$\qquad\qquad$ *if* $\quad$ Dest(msg) = H

$\rightarrow$ Sys(Msite(mem, $min \odot msg$, $mout$), Site(id, $cache$, cin, $cout$, pmb, mpb, $proc$) | $sites$)

Message-Mem-to-Cache Rule

$\quad$ Sys(Msite(mem, min, $msg \otimes mout$), Site(id, $cache$, cin, $cout$, pmb, mpb, $proc$) | $sites$)

$\qquad\qquad$ *if* $\quad$ Dest(msg) = id

$\rightarrow$ Sys(Msite(mem, min, $mout$), Site(id, $cache$, $cin \odot msg$, $cout$, pmb, mpb, $proc$) | $sites$)

Non-FIFO vs. FIFO Message Passing We use outgoing queues to model non-FIFO and FIFO networks. Non-FIFO message passing provides no guarantee on the order in which messages are delivered. This can be effectively characterized by allowing messages to be reordered arbitrarily in an outgoing queue (thus any outgoing message can be brought to the front of the queue). FIFO message passing, on the other hand, allows outgoing messages to be reordered only when they have different destinations or addresses. It is worth emphasizing that FIFO message passing does not guarantee the ordering for messages with different addresses, even if they have the same source and destination. The rational behind is that memory cells of different addresses may physically reside in different sites in DSM systems (that is, the home 'H' may represent more than one site in the implementation).

$$\begin{array}{lccl}
\text{Non-FIFO:} & msg_1 \otimes msg_2 & \equiv & msg_2 \otimes msg_1 \\
\text{FIFO:} & msg_1 \otimes msg_2 & \equiv & msg_2 \otimes msg_1 \\
 & \multicolumn{3}{r}{\textit{if}\quad \mathrm{Dest}(msg_1) \neq \mathrm{Dest}(msg_2) \;\vee\; \mathrm{Addr}(msg_1) \neq \mathrm{Addr}(msg_2)}
\end{array}$$

The message passing rules are mandatory rules in the sense that each outgoing message must be delivered to its destination in finite time. Unless otherwise specified, FIFO message passing is assumed for all the protocols described in this thesis.

Buffer Management Ideally we would like to treat incoming queues as FIFOs and process incoming messages in the order in which they are received. However, this may cause deadlock or livelock unless messages that cannot be processed temporarily are properly buffered so that following messages can still be processed. Conceptually, this can be modeled by allowing incoming messages to be reordered under appropriate conditions so that stalled messages do not block other messages.

Different protocols have different buffering requirements that can be specified by different reordering conditions. For example, by allowing incoming messages with different sources or addresses to be reordered arbitrarily, we can treat each incoming queue as a set of FIFO subqueues. This prevents messages with different sources or addresses to block each other.

Sub-queues for different sources or addresses:

$$msg_1 \odot msg_2 \quad \equiv \quad msg_2 \odot msg_1$$
$$\textit{if}\quad \mathrm{Src}(msg_1) \neq \mathrm{Src}(msg_2) \;\vee\; \mathrm{Addr}(msg_1) \neq \mathrm{Addr}(msg_2)$$

Exposing buffer management at early design stages is inappropriate, since it could give rise to a bloated set of rules and dramatically complicate the protocol verification. Instead, we first model buffer management via message reordering in incoming queues, and then determine what must be done in practice in order to satisfy the reordering requirements. This strategy can lead to more concise specifications and simplify protocol design and verification.

We use outgoing queues to model FIFO and non-FIFO networks, and incoming queues to model message buffering at destinations. For outgoing queues, it is desirable to have less stringent reordering conditions since this implies less requirements on message ordering that must be guaranteed by the network. For incoming queues, it is desirable to have more stringent reordering conditions since this implies less effort for message buffering.

Notations Given a system term s, we use Mem(s), Min(s) and Mout(s) to represent the memory, the memory's incoming queue and the memory's outgoing queue, respectively. For a cache site id, we use $\mathrm{Cache}_{id}(s)$, $\mathrm{Cin}_{id}(s)$, $\mathrm{Cout}_{id}(s)$, $\mathrm{Pmb}_{id}(s)$, $\mathrm{Mpb}_{id}(s)$ and $\mathrm{Proc}_{id}(s)$ to represent the cache, the incoming queue, the outgoing queue, the processor-to-memory buffer, the memory-to-processor buffer and the processor, respectively. More precisely, let s be the term

$$\mathrm{Sys}(\mathrm{Msite}(\mathit{mem}, \mathit{min}, \mathit{mout}), \mathrm{Site}(id, \mathit{cache}, \mathit{cin}, \mathit{cout}, \mathit{pmb}, \mathit{mpb}, \mathit{proc}) \mid \mathit{sites})$$

We define the following notations:

$$\begin{array}{lcl}
\mathrm{Mem}(s) & \equiv & \mathit{mem} \\
\mathrm{Min}(s) & \equiv & \mathit{min} \\
\mathrm{Mout}(s) & \equiv & \mathit{mout} \\
\mathrm{Cache}_{id}(s) & \equiv & \mathit{cache} \\
\mathrm{Cin}_{id}(s) & \equiv & \mathit{cin} \\
\mathrm{Cout}_{id}(s) & \equiv & \mathit{cout} \\
\mathrm{Pmb}_{id}(s) & \equiv & \mathit{pmb} \\
\mathrm{Mpb}_{id}(s) & \equiv & \mathit{mpb} \\
\mathrm{Proc}_{id}(s) & \equiv & \mathit{proc}
\end{array}$$

We say a message is in transit from a cache to the memory if the message is in the cache's outgoing queue or the memory's incoming queue. We say a message is in transit from the memory to a cache if the message is in the memory's outgoing queue or the cache's incoming queue. We use shorthand notation $\mathrm{MoutCin}_{id}(s)$ to represent the messages in transit from the memory to cache site id, and $\mathrm{MinCout}_{id}(s)$ the messages in transit from cache site id to the memory. That is:

$$\begin{array}{lll}
\mathit{msg} \in \mathrm{MoutCin}_{id}(s) & \mathit{iff} & \mathit{msg} \in \mathrm{Mout}(s) \;\vee\; \mathit{msg} \in \mathrm{Cin}_{id}(s) \\
\mathit{msg} \in \mathrm{MinCout}_{id}(s) & \mathit{iff} & \mathit{msg} \in \mathrm{Min}(s) \;\vee\; \mathit{msg} \in \mathrm{Cout}_{id}(s)
\end{array}$$

Let msg_1 and msg_2 be messages between the same source and destination regarding the same address. We say message msg_1 precedes message msg_2 (or msg_2 follows msg_1), if the following condition is satisfied:

- msg_1 and msg_2 are both in the source's outgoing queue, and msg_1 is in front of message msg_2; or
- msg_1 and msg_2 are both in the destination's incoming queue, and msg_1 is in front of message msg_2; or
- msg_1 is in the destination's incoming queue and msg_2 is in the source's outgoing queue.

We use notation $\mathit{msg}_{\uparrow}$ to represent that message msg has no preceding message, and notation $\mathit{msg}_{\downarrow}$ to represent that message msg has no following message. Notation $\mathit{msg}_{\updownarrow}$ means that message msg has no preceding or following message (that is, it is the only message regarding the address between the source and destination).

4.3 The Imperative Rules of the Base Protocol

The Base protocol is a simple implementation of the CRF model. It is a directory-less protocol in the sense that the memory maintains no directory information about where an address is cached. In Base, the memory behaves as the rendezvous: when a processor executes a Commit on a dirty cell, it writes the data back to the memory before completing the Commit; when a processor executes a Reconcile on a clean cell, it purges the data before completing the Reconcile (thus a following Loadl to the same address must retrieve data from the memory).

The Base protocol employs two stable cache states, Clean and Dirty, and two transient cache states, CachePending and WbPending. When an address is not resident in a cache, we say the cache state is Invalid or the address is cached in the Invalid state. There are four messages, CacheReq, Cache, Wb and WbAck.

The imperative rules of Base specify how instructions can be executed on cache cells in appropriate states and how data can be propagated between the memory and caches. They are responsible for ensuring the soundness of the protocol, that is, the system only exhibits behaviors that are permitted by the CRF model. The imperative rules have three sets of rules, the processor rules, the cache engine rules and the memory engine rules.

Processor Rules: A Loadl or Storel instruction can be performed if the address is cached in the Clean or Dirty state. A Commit instruction can be performed if the address is uncached or cached in the Clean state. A Reconcile instruction can be performed if the address is uncached or cached in the Dirty state.

Loadl-on-Clean Rule

$\quad\;\;\text{Site}(id,\ \text{Cell}(a,v,\text{Clean})\,|\,cache,\ in,\ out,\ \langle t,\text{Loadl}(a)\rangle;pmb,\ mpb,\ proc)$

$\to\ \text{Site}(id,\ \text{Cell}(a,v,\text{Clean})\,|\,cache,\ in,\ out,\ pmb,\ mpb|\langle t,v\rangle,\ proc)$

Loadl-on-Dirty Rule

$\quad\;\;\text{Site}(id,\ \text{Cell}(a,v,\text{Dirty})\,|\,cache,\ in,\ out,\ \langle t,\text{Loadl}(a)\rangle;pmb,\ mpb,\ proc)$

$\to\ \text{Site}(id,\ \text{Cell}(a,v,\text{Dirty})\,|\,cache,\ in,\ out,\ pmb,\ mpb|\langle t,v\rangle,\ proc)$

Storel-on-Clean Rule

$\quad\;\;\text{Site}(id,\ \text{Cell}(a,\text{-},\text{Clean})\,|\,cache,\ in,\ out,\ \langle t,\text{Storel}(a,v)\rangle;pmb,\ mpb,\ proc)$

$\to\ \text{Site}(id,\ \text{Cell}(a,v,\text{Dirty})\,|\,cache,\ in,\ out,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc)$

Storel-on-Dirty Rule

$\quad\;\;\text{Site}(id,\ \text{Cell}(a,\text{-},\text{Dirty})\,|\,cache,\ in,\ out,\ \langle t,\text{Storel}(a,v)\rangle;pmb,\ mpb,\ proc)$

$\to\ \text{Site}(id,\ \text{Cell}(a,v,\text{Dirty})\,|\,cache,\ in,\ out,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc)$

Commit-on-Clean Rule

$\quad\;\;\text{Site}(id,\ \text{Cell}(a,v,\text{Clean})\,|\,cache,\ in,\ out,\ \langle t,\text{Commit}(a)\rangle;pmb,\ mpb,\ proc)$

$\to\ \text{Site}(id,\ \text{Cell}(a,v,\text{Clean})\,|\,cache,\ in,\ out,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc)$

Commit-on-Invalid Rule

$\quad\;\;\text{Site}(id,\ cache,\ in,\ out,\ \langle t,\text{Commit}(a)\rangle;pmb,\ mpb,\ proc) \qquad \textit{if} \quad a \notin cache$

$\to\ \text{Site}(id,\ cache,\ in,\ out,\ pmb,\ mpb|\langle t,\text{Ack}\rangle,\ proc)$

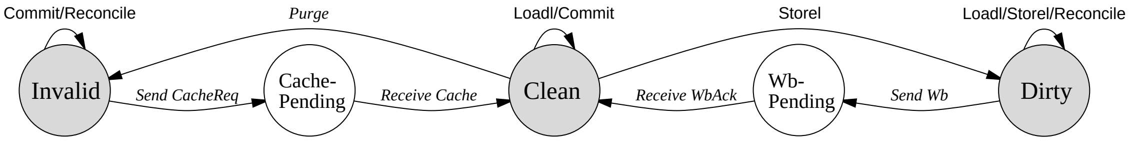


Figure 4.3: Cache State Transitions of Base

Reconcile-on-Dirty Rule

$\quad$ Site(id, Cell(a,v,Dirty) | $cache$, in, out, $\langle t$,Reconcile(a)$\rangle$;pmb, mpb, $proc$)

→ $\quad$ Site(id, Cell(a,v,Dirty) | $cache$, in, out, pmb, $mpb|\langle t$,Ack$\rangle$, $proc$)

Reconcile-on-Invalid Rule

$\quad$ Site(id, $cache$, in, out, $\langle t$,Reconcile(a)$\rangle$;pmb, mpb, $proc$) $\qquad$ *if* $\quad a \notin cache$

→ $\quad$ Site(id, $cache$, in, out, pmb, $mpb|\langle t$,Ack$\rangle$, $proc$)

C-engine Rules: A cache can purge a clean cell at any time. It can also write the data of a dirty cell to the memory via a writeback message (Wb), and set the cache state to WbPending, indicating that a writeback operation is being performed on the address. The cache state becomes Clean when a writeback acknowledgment (WbAck) is received.

For an uncached address, a cache can send a cache request (CacheReq) to the memory to request the data, and set the cache state to CachePending, indicating that a cache copy is being requested on the address. When a cache message (Cache) is received, the data is written to the cache cell and the cache state becomes Clean. Figure 4.3 shows the cache state transitions.

C-Purge Rule

$\quad$ Site(id, Cell(a,-,Clean) | $cache$, in, out, pmb, mpb, $proc$)

→ $\quad$ Site(id, $cache$, in, out, pmb, mpb, $proc$)

C-Send-Wb Rule

$\quad$ Site(id, Cell(a,v,Dirty) | $cache$, in, out, pmb, mpb, $proc$)

→ $\quad$ Site(id, Cell(a,v,WbPending) | $cache$, in, $out \otimes$ Msg(id,H,Wb,a,v), pmb, mpb, $proc$)

C-Send-CacheReq Rule

$\quad$ Site(id, $cache$, in, out, pmb, mpb, $proc$) $\qquad$ *if* $\quad a \notin cache$

→ $\quad$ Site(id, Cell(a,-,CachePending) | $cache$, in, $out \otimes$ Msg(id,H,CacheReq,a), pmb, mpb, $proc$)

C-Receive-Cache Rule

$\quad$ Site(id, Cell(a,-,CachePending) | $cache$, Msg(H,id,Cache,a,v) $\odot$ in , out, pmb, mpb, $proc$)

→ $\quad$ Site(id, Cell(a,v,Clean) | $cache$, in, out, pmb, mpb, $proc$)

C-Receive-WbAck Rule

$\quad$ Site(id, Cell(a,v,WbPending) | $cache$, Msg(H,id,WbAck,a) $\odot$ in , out, pmb, mpb, $proc$)

→ $\quad$ Site(id, Cell(a,v,Clean) | $cache$, in, out, pmb, mpb, $proc$)

Imperative Processor Rules

Instruction	Cstate	Action	Next Cstate	
Loadl(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	IP1
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	IP2
Storel(a,v)	Cell(a,-,Clean)	*retire*	Cell(a,v,Dirty)	IP3
	Cell(a,-,Dirty)	*retire*	Cell(a,v,Dirty)	IP4
Commit(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	IP5
	$a \notin cache$	*retire*	$a \notin cache$	IP6
Reconcile(a)	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	IP7
	$a \notin cache$	*retire*	$a \notin cache$	IP8

Imperative C-engine Rules

Msg from H	Cstate	Action	Next Cstate	
	Cell(a,-,Clean)		$a \notin cache$	IC1
	Cell(a,v,Dirty)	$\langle$Wb,a,$v\rangle \rightarrow$ H	Cell(a,v,WbPending)	IC2
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ H	Cell(a,-,CachePending)	IC3
$\langle$Cache,a,$v\rangle$	Cell(a,-,CachePending)		Cell(a,v,Clean)	IC4
$\langle$WbAck,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean)	IC5

Imperative M-engine Rules

Msg from id	Mstate	Action	Next Mstate	
$\langle$CacheReq,$a\rangle$	Cell(a,v)	$\langle$Cache,a,$v\rangle \rightarrow id$	Cell(a,v)	IM1
$\langle$Wb,a,$v\rangle$	Cell(a,-)	$\langle$WbAck,$a\rangle \rightarrow id$	Cell(a,v)	IM2

Figure 4.4: Imperative Rules of Base

M-engine Rules: The memory handles cache requests and writeback messages. When a cache request is received, the memory sends a cache message to supply the requested data to the requesting site. When a writeback message is received, the memory updates the memory with the committed data and sends an acknowledgment back to the cache.

M-Receive-CacheReq-&-Send-Cache Rule

Msite(Cell(a,v) | mem, Msg(id,H,CacheReq,a) $\odot$ in , out)

$\rightarrow$ Msite(Cell(a,v) | mem, in, $out \otimes$ Msg(H,id,Cache,a,v))

M-Receive-Wb-&-Send-WbAck Rule

Msite(Cell(a,-) | mem, Msg(id,H,Wb,a,v) $\odot$ in , out)

$\rightarrow$ Msite(Cell(a,v) | mem, in, $out \otimes$ Msg(H,id,WbAck,a))

Figure 4.4 summarizes the imperative rules of Base. The shorthand notation '$\langle cmd,a,v\rangle$' represents a message with command cmd, address a and value v. The source and destination are implicit: for cache engine rules, an incoming message's source and destination are the memory (H) and cache id, respectively; for memory engine rules, an incoming message's source and destination are cache id and the memory (H). The notation '$msg \rightarrow dest$' means sending the message msg to the destination $dest$.

In the tabular description, when a processor completes or retires an instruction, the instruction is removed from the processor-to-memory buffer and the corresponding data or acknowledgment is sent to the memory-to-processor buffer. Similarly, when a cache or memory engine receives and processes an incoming message, the message is consumed and removed from the incoming queue.

Mandatory Processor Rules					
Instruction	**Cstate**	**Action**	**Next Cstate**		
Loadl(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	P1	SF
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	P2	SF
	Cell(a,v,WbPending)	*stall*	Cell(a,v,WbPending)	P3	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P4	
	$a \notin$ *cache*	*stall*, $\langle\text{CacheReq},a\rangle \rightarrow \text{H}$	Cell(a,-,CachePending)	P5	
Storel(a,v)	Cell(a,-,Clean)	*retire*	Cell(a,v,Dirty)	P6	SF
	Cell(a,-,Dirty)	*retire*	Cell(a,v,Dirty)	P7	SF
	Cell(a,v_1,WbPending)	*stall*	Cell(a,v_1,WbPending)	P8	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P9	
	$a \notin$ *cache*	*stall*, $\langle\text{CacheReq},a\rangle \rightarrow \text{H}$	Cell(a,-,CachePending)	P10	
Commit(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	P11	SF
	Cell(a,v,Dirty)	*stall*, $\langle\text{Wb},a,v\rangle \rightarrow \text{H}$	Cell(a,v,WbPending)	P12	
	Cell(a,v,WbPending)	*stall*	Cell(a,v,WbPending)	P13	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P14	
	$a \notin$ *cache*	*retire*	$a \notin$ *cache*	P15	SF
Reconcile(a)	Cell(a,-,Clean)	*stall*	$a \notin$ *cache*	P16	
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	P17	SF
	Cell(a,v,WbPending)	*stall*	Cell(a,v,WbPending)	P18	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P19	
	$a \notin$ *cache*	*retire*	$a \notin$ *cache*	P20	SF

Voluntary C-engine Rules				
	Cstate	**Action**	**Next Cstate**	
	Cell(a,-,Clean)		$a \notin$ *cache*	VC1
	Cell(a,v,Dirty)	$\langle\text{Wb},a,v\rangle \rightarrow \text{H}$	Cell(a,v,WbPending)	VC2
	$a \notin$ *cache*	$\langle\text{CacheReq},a\rangle \rightarrow \text{H}$	Cell(a,-,CachePending)	VC3

Mandatory C-engine Rules				
Msg from H	**Cstate**	**Action**	**Next Cstate**	
$\langle\text{Cache},a,v\rangle$	Cell(a,-,CachePending)		Cell(a,v,Clean)	MC1
$\langle\text{WbAck},a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean)	MC2

Mandatory M-engine Rules				
Msg from *id*	**Mstate**	**Action**	**Next Mstate**	
$\langle\text{CacheReq},a\rangle$	Cell(a,v)	$\langle\text{Cache},a,v\rangle \rightarrow id$	Cell(a,v)	MM1
$\langle\text{Wb},a,v\rangle$	Cell(a,-)	$\langle\text{WbAck},a\rangle \rightarrow id$	Cell(a,v)	MM2

Figure 4.5: The Base Protocol

4.4 The Base Protocol

The imperative rules rely on an oracle to invoke appropriate coherence actions at appropriate times. In Base, an instruction can be stalled if the address is not cached in an appropriate state. This happens, for example, when a processor intends to execute a Loadl or Storel instruction on an address that is not cached. To ensure liveness, the cache engine must take proper actions to ensure that a stalled instruction will be completed eventually. There are three cases:

- When a Loadl or Storel is stalled because of a cache miss, the cache sends a cache request to the memory to request the data.
- When a Commit is stalled on a dirty cell, the cache writes the data back to the memory.
- When a Reconcile is stalled on a clean cell, the cache purges the clean copy (a potential optimization may allow the Reconcile to be retired at the same time).

Figure 4.5 defines the rules of the Base protocol. The informal tabular description can be easily translated into formal TRS rules (cases that are not specified represent illegal states). The processor rules and the memory engine rules are all mandatory rules, and the cache engine rules are further classified into voluntary and mandatory rules. Weak fairness is guaranteed for each mandatory rule, that is, an enabled mandatory rule will be applied eventually or become impossible to be applied at some time. To ensure liveness, certain mandatory rules need to be strongly fair and are marked with 'SF' in the table. When a strongly fair rule can be applied, it will be applied eventually or become impossible to be applied forever.

A processor rule retires or stalls an instruction depending on the cache state of the accessed address. When an instruction is retired, it is removed from the processor-to-memory buffer and the requested data or acknowledgment is supplied to the memory-to-processor buffer. When an instruction is stalled, it remains in the processor-to-memory buffer for later processing. Certain processor rules cause no action; this means the stalled instruction may need to be retried in practice in order to be completed. Note that a stalled instruction does not necessarily block following instructions to be executed, since the processor can reorder instructions in the processor-to-memory buffer according to CRF reordering rules.

When a cache or memory engine receives a message, the message is immediately removed from the incoming queue once it is processed. In Base, no message needs to be stalled since an incoming message can always be processed when it is received.

Voluntary rules: A cache can purge a clean cell at any time. It can write the data of a dirty cell back to the memory via a writeback message. A cache can send a cache request to the memory to request the data if the address is not cached. One subtle issue worth noting is that it is not safe for the memory to send data to a cache that has not requested the data.

Voluntary rules can be used to improve the system performance without compromising the soundness and liveness properties of the protocol. For example, a cache can evict a dirty cell by writing back and purging if it decides that the data is unlikely to be accessed later. This can potentially accelerate the execution of future Commit instructions on the same address. Similarly, a cache can prefetch data by issuing a cache-request message.

Optimization: In Base, an instruction is stalled when the address is cached in a transient state. This constraint can be relaxed under certain circumstances:

- A Loadl instruction can complete if the address is cached in the WbPending state; the cache state remains unchanged.
- A Storel instruction can complete in case of a cache miss; a dirty cell with the stored data is added into the cache.
- A Commit instruction can complete if the address is cached in the CachePending state; the cache state remains unchanged.

4.5 Soundness Proof of the Base Protocol

Soundness of a cache coherence protocol means that the TRS specifying the protocol can be simulated by the TRS specifying the memory model. In this section, we prove the soundness of the Base protocol by showing that CRF can simulate Base. We define a mapping function from Base to CRF, and show that any imperative rule of Base can be simulated in CRF with respect to the mapping function. The soundness of the Base protocol follows from the fact that all the Base rules can be derived from the Base imperative rules.

Before delving into the proof, we present some invariants that will be used throughout the proof. The invariants also help us understand the behavior of the protocol.

4.5.1 Some Invariants of Base

Lemma 1 consists of two invariants that describe the correspondence between cache states and messages in transit. An address is cached in the CachePending state, if and only if there is a CacheReq or Cache message between the cache and the memory. An address is cached in the WbPending state, if and only if there is a Wb or WbAck message between the cache and the memory.

Lemma 1 Given a Base term s,

(1) $\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(s) \quad \Leftrightarrow$
$$\text{Msg}(id,\text{H},\text{CacheReq},a) \in \text{MinCout}_{id}(s) \;\vee\; \text{Msg}(\text{H},id,\text{Cache},a,\text{-}) \in \text{MoutCin}_{id}(s)$$

(2) $\text{Cell}(a,v,\text{WbPending}) \in \text{Cache}_{id}(s) \quad \Leftrightarrow$
$$\text{Msg}(id,\text{H},\text{Wb},a,v) \in \text{MinCout}_{id}(s) \;\vee\; \text{Msg}(\text{H},id,\text{WbAck},a) \in \text{MoutCin}_{id}(s)$$

Proof The proof is based on induction on rewriting steps. The invariants hold trivially for the initial term where all caches and queues are empty. It can be shown by checking each rule that, if the invariants hold for a term, they still hold after the term is rewritten according to that rule. □

Lemma 2 means that there exists at most one message in transit between the same source and destination regarding the same address. This implies that the soundness and liveness of the Base protocol are not contingent upon the FIFO order of message passing.

Lemma 2 Given a Base term s,

$msg_1 \in s \;\wedge\; msg_2 \in s \quad \Rightarrow$
$$\text{Src}(msg_1) \neq \text{Src}(msg_2) \;\vee\; \text{Dest}(msg_1) \neq \text{Dest}(msg_2) \;\vee\; \text{Addr}(msg_1) \neq \text{Addr}(msg_2)$$

Proof The proof is based on induction on rewriting steps. The invariant holds trivially for the initial term where all queues are empty. It can be shown by checking each rule that, if the invariant holds for a term, it still holds after the term is rewritten according to that rule. Note that a cache can send a message only when the address is uncached or cached in a stable state (which means there is no message between the cache and the memory regarding the address), while the memory can send a message only when it receives an incoming message. □

4.5.2 Mapping from Base to CRF

We define a mapping function that maps terms of Base to terms of CRF. For Base terms in which the message queues are all empty, it is straightforward to find the corresponding CRF terms. We call such Base terms drained terms. There is a one-to-one correspondence between the drained terms of Base and the terms of CRF. For Base terms that contain non-empty message queues, we apply a set of draining rules so that all the messages in transit will be removed from the queues eventually.

The intuition behind the draining rules is that message queues can always be drained via forward or backward draining. With forward draining, we can move a message to its destination and consume the message at the destination; with backward draining, we can move a message back to its source and reclaim the message at the source. While forward draining is preferred since it can be achieved by employing some existing rules, backward draining is needed when forward draining would lead to non-deterministic results. This can happen, for example, when multiple Wb messages (from different caches) regarding the same address are in transit to the memory.

There are many different ways to drain messages from the network. We use forward draining for messages from the memory to caches, and backward draining for messages from caches to the memory. For example, to drain a Cache message in the memory's outgoing queue, we first pass the message to the destination's incoming queue, and then cache the data in the cache. To drain a Wb message in the memory's incoming queue, we first pass the message back to the source's outgoing queue and then restore the cache state.

Backward Rules: The *Backward-Message-Cache-to-Mem* rule passes a message from the incoming queue of the memory back to the outgoing queue of the source cache. The *Backward-C-Send-Wb* and *Backward-C-Send-CacheReq* rules allow a cache to retrieve Wb and CacheReq messages from the network and recover corresponding cache cells.

Backward-Message-Cache-to-Mem Rule

$$\mathrm{Sys}(\mathrm{Msite}(\mathit{mem},\ \mathit{min} \odot \mathit{msg},\ \mathit{mout}),\ \mathrm{Site}(\mathit{id},\ \mathit{cache},\ \mathit{cin},\ \mathit{cout},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc}) \mid \mathit{sites})$$

$$\textit{if}\quad \mathrm{Src}(\mathit{msg}) = \mathit{id}$$

$$\rightarrow\quad \mathrm{Sys}(\mathrm{Msite}(\mathit{mem},\ \mathit{min},\ \mathit{mout}),\ \mathrm{Site}(\mathit{id},\ \mathit{cache},\ \mathit{cin},\ \mathit{msg} \otimes \mathit{cout},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc}) \mid \mathit{sites})$$

Backward-C-Send-Wb Rule

$$\mathrm{Site}(\mathit{id},\ \mathrm{Cell}(a,v,\mathrm{WbPending}) \mid \mathit{cache},\ \mathit{in},\ \mathit{out} \otimes \mathrm{Msg}(\mathit{id},\mathrm{H},\mathrm{Wb},a,v),\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc})$$

$$\rightarrow\quad \mathrm{Site}(\mathit{id},\ \mathrm{Cell}(a,v,\mathrm{Dirty}) \mid \mathit{cache},\ \mathit{in},\ \mathit{out},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc})$$

Backward-C-Send-CacheReq Rule

$$\mathrm{Site}(\mathit{id},\ \mathrm{Cell}(a,\text{-},\mathrm{CachePending}) \mid \mathit{cache},\ \mathit{in},\ \mathit{out} \otimes \mathrm{Msg}(\mathit{id},\mathrm{H},\mathrm{CacheReq},a),\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc})$$

$$\rightarrow\quad \mathrm{Site}(\mathit{id},\ \mathit{cache},\ \mathit{in},\ \mathit{out},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc})$$

The backward rules above are the backward version of the *Message-Cache-to-Mem*, *C-Send-Wb* and *C-Send-CacheReq* rules, and can be used to drain the Wb and CacheReq messages. It is trivial to show that Lemmas 1 and 2 still hold in the presence of the backward rules.

The draining rules consist of some Base rules that are needed to drain Cache and WbAck messages, and the backward rules that are needed to drain CacheReq and Wb messages.

Definition 3 (Draining Rules) Given a Base term s, the corresponding drained term $\text{dr}(s)$ is the normal form of s with respect to the following draining rules:

$$\begin{array}{lll} D & \equiv & \{ \quad \textit{C-Receive-Cache}, \textit{Backward-C-Send-CacheReq}, \\ & & \quad\;\; \textit{C-Receive-WbAck}, \textit{Backward-C-Send-Wb}, \\ & & \quad\;\; \textit{Message-Mem-to-Cache}, \textit{Backward-Message-Cache-to-Mem} \quad \} \end{array}$$

Lemma 4 D is strongly terminating and confluent, that is, rewriting a Base term with respect to the draining rules always terminates and reaches the same normal form, regardless of the order in which the rules are applied.

Proof The termination is obvious because according to the draining rules, messages can only flow from memory to caches and will be consumed at caches. The confluence follows from the fact that the draining rules do not interfere with each other. □

Lemma 5 states that the memory, processor-to-memory buffers, memory-to-processor buffers and processors all remain unchanged while a term is drained. Lemma 6 states that the incoming and outgoing queues all become empty in a drained term. This follows from Lemma 1 which guarantees that an incoming message can always be consumed.

Lemma 5 Given a Base term s,

(1) $\text{Mem}(s) = \text{Mem}(\text{dr}(s))$
(2) $\text{Pmb}_{id}(s) = \text{Pmb}_{id}(\text{dr}(s))$
(3) $\text{Mpb}_{id}(s) = \text{Mpb}_{id}(\text{dr}(s))$
(4) $\text{Proc}_{id}(s) = \text{Proc}_{id}(\text{dr}(s))$

Lemma 6 Given a Base term s,

(1) $\text{Min}(\text{dr}(s)) = \epsilon$
(2) $\text{Mout}(\text{dr}(s)) = \epsilon$
(3) $\text{Cin}_{id}(\text{dr}(s)) = \epsilon$
(4) $\text{Cout}_{id}(\text{dr}(s)) = \epsilon$

Lemma 7 ensures that a stable cache cell remains unaffected when a term is drained. Lemma 8 ensures that if a term contains a message, the cache cell will be in an appropriate stable state in the drained term. Lemma 7 is obvious since the draining rules do not modify stable cache cells. Lemma 8 follows from Lemma 1 and the draining rules (note that messages can be drained in any order because of the confluence of the draining rules).

Lemma 7 Given a Base term s,

(1) $\text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(s) \Rightarrow \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s))$
(2) $\text{Cell}(a,v,\text{Dirty}) \in \text{Cache}_{id}(s) \Rightarrow \text{Cell}(a,v,\text{Dirty}) \in \text{Cache}_{id}(\text{dr}(s))$
(3) $a \notin \text{Cache}_{id}(s) \Rightarrow a \notin \text{Cache}_{id}(\text{dr}(s))$

Base Imperative Rule	**CRF Rule**
IP1 (*Loadl-on-Clean*)	*CRF-Loadl*
IP2 (*Loadl-on-Dirty*)	*CRF-Loadl*
IP3 (*Storel-on-Clean*)	*CRF-Storel*
IP4 (*Storel-on-Dirty*)	*CRF-Storel*
IP5 (*Commit-on-Clean*)	*CRF-Commit*
IP6 (*Commit-on-Invalid*)	*CRF-Commit*
IP7 (*Reconcile-on-Dirty*)	*CRF-Reconcile*
IP8 (*Reconcile-on-Invalid*)	*CRF-Reconcile*
IC1 (*C-Purge*)	*CRF-Purge*
IC2 (*C-Send-Wb*)	ϵ
IC3 (*C-Send-CacheReq*)	ϵ
IC4 (*C-Receive-Cache*)	ϵ
IC5 (*C-Receive-WbAck*)	ϵ
IM1 (*M-Receive-CacheReq-&-Send-Cache*)	*CRF-Cache*
IM2 (*M-Receive-Wb-&-Send-WbAck*)	*CRF-Writeback*
Message-Mem-to-Cache	ϵ
Message-Cache-to-Mem	ϵ

Figure 4.6: Simulation of Base in CRF

Lemma 8 Given a Base term s,

(1) $\text{Msg}(id,\text{H},\text{CacheReq},a) \in \text{MinCout}_{id}(s) \quad \Rightarrow \quad a \notin \text{Cache}_{id}(\text{dr}(s))$

(2) $\text{Msg}(\text{H},id,\text{Cache},a,v) \in \text{MoutCin}_{id}(s) \quad \Rightarrow \quad \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s))$

(3) $\text{Msg}(id,\text{H},\text{Wb},a,v) \in \text{MinCout}_{id}(s) \quad \Rightarrow \quad \text{Cell}(a,v,\text{Dirty}) \in \text{Cache}_{id}(\text{dr}(s))$

(4) $\text{Msg}(\text{H},id,\text{WbAck},a) \in \text{MoutCin}_{id}(s) \quad \Rightarrow \quad \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s))$

Definition 9 (Mapping from Base to CRF) Given a Base term s, the corresponding CRF term $\text{f}(s)$ is the drained term $\text{dr}(s)$ with message queues and cache identifiers removed.

It is obvious that the mapping function maps the initial Base term (with all caches and network queues empty) to the initial CRF term (with all semantic caches empty). Furthermore, the mapping function guarantees that any Base term is mapped to a legal CRF term (this will follow trivially from the simulation theorem given below).

4.5.3 Simulation of Base in CRF

Theorem 10 (CRF Simulates Base) Given Base terms s_1 and s_2,

$$s_1 \rightarrow s_2 \quad \text{in Base} \qquad \Rightarrow \qquad \text{f}(s_1) \twoheadrightarrow \text{f}(s_2) \quad \text{in CRF}$$

Proof The proof is based on a case analysis on the imperative rule used in the rewriting of '$s_1 \rightarrow s_2$' in Base. Let a and id be the address and the cache identifier, respectively.

Imperative Processor Rules

- If Rule IP1 (*Loadl-on-Clean*) applies, then

$$\begin{array}{ll} & \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(s_1) \ \wedge \ \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(s_2) \\ \Rightarrow & \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s_1)) \ \wedge \ \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s_2)) \quad (\textit{Lemma 7}) \\ \Rightarrow & \text{f}(s_1) \rightarrow \text{f}(s_2) \qquad (\textit{CRF-Loadl}) \end{array}$$

- If Rule IP2 (*Loadl-on-Dirty*) applies, then

$$\begin{aligned}
& \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_1) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_2) \\
\Rightarrow\; & \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 7}) \\
\Rightarrow\; & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Loadl})
\end{aligned}$$

- If Rule IP3 (*Storel-on-Clean*) applies, then

$$\begin{aligned}
& \mathrm{Cell}(a,\text{-},\mathrm{Clean}) \in \mathrm{Cache}_{id}(s_1) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_2) \\
\Rightarrow\; & \mathrm{Cell}(a,\text{-},\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 7}) \\
\Rightarrow\; & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Storel})
\end{aligned}$$

- If Rule IP4 (*Storel-on-Dirty*) applies, then

$$\begin{aligned}
& \mathrm{Cell}(a,\text{-},\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_1) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_2) \\
\Rightarrow\; & \mathrm{Cell}(a,\text{-},\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 7}) \\
\Rightarrow\; & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Storel})
\end{aligned}$$

- If Rule IP5 (*Commit-on-Clean*) applies, then

$$\begin{aligned}
& \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(s_1) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(s_2) \\
\Rightarrow\; & \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 7}) \\
\Rightarrow\; & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Commit})
\end{aligned}$$

- If Rule IP6 (*Commit-on-Invalid*) applies, then

$$\begin{aligned}
& a \notin \mathrm{Cache}_{id}(s_1) \;\wedge a \notin \mathrm{Cache}_{id}(s_2) \\
\Rightarrow\; & a \notin \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; a \notin \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 7}) \\
\Rightarrow\; & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Commit})
\end{aligned}$$

- If Rule IP7 (*Reconcile-on-Dirty*) applies, then

$$\begin{aligned}
& \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_1) \;\wedge \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_2) \\
\Rightarrow\; & \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 7}) \\
\Rightarrow\; & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Reconcile})
\end{aligned}$$

- If Rule IP8 (*Reconcile-on-Invalid*) applies, then

$$\begin{aligned}
& a \notin \mathrm{Cache}_{id}(s_1) \;\wedge\; a \notin \mathrm{Cache}_{id}(s_2) \\
\Rightarrow\; & a \notin \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; a \notin \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 7}) \\
\Rightarrow\; & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Reconcile})
\end{aligned}$$

Imperative C-engine Rules

- If Rule IC1 (*C-Purge*) applies, then

$$\begin{aligned}
& \mathrm{Cell}(a,\text{-},\mathrm{Clean}) \in \mathrm{Cache}_{id}(s_1) \;\wedge\; a \notin \mathrm{Cache}_{id}(s_2) \\
\Rightarrow\; & \mathrm{Cell}(a,\text{-},\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; a \notin \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 7}) \\
\Rightarrow\; & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Purge})
\end{aligned}$$

Imperative M-engine Rules

Base Rule	**Base Imperative Rule**
P1	IP1 (*Loadl-on-Clean*)
P2	IP2 (*Loadl-on-Dirty*)
P3	ϵ
P4	ϵ
P5	IC3 (*C-Send-CacheReq*)
P6	IP3 (*Storel-on-Clean*)
P7	IP4 (*Storel-on-Dirty*)
P8	ϵ
P9	ϵ
P10	IC3 (*C-Send-CacheReq*)
P11	IP5 (*Commit-on-Clean*)
P12	IC2 (*C-Send-Wb*)
P13	ϵ
P14	ϵ
P15	IP6 (*Commit-on-Invalid*)
P16	IC1 (*C-Purge*)
P17	IP7 (*Reconcile-on-Dirty*)
P18	ϵ
P19	ϵ
P20	IP8 (*Reconcile-on-Invalid*)
VC1	IC1 (*C-Purge*)
VC2	IC2 (*C-Send-Wb*)
VC3	IC3 (*C-Send-CacheReq*)
MC1	IC4 (*C-Receive-Cache*)
MC2	IC5 (*C-Receive-WbAck*)
MM1	IM1 (*M-Receive-CacheReq-&-Send-Cache*)
MM2	IM2 (*M-Receive-Wb-&-Send-WbAck*)

Figure 4.7: Derivation of Base from Imperative Rules

- If Rule IM1 (*M-Receive-CacheReq-&-Send-Cache*) applies, then

$$\begin{array}{lll}
 & \text{Cell}(a,v) \in \text{Mem}(s_1) \ \wedge\ \text{Msg}(id,\text{H},\text{CacheReq},a) \in \text{Min}(s_1) \ \wedge & \\
 & \text{Cell}(a,v) \in \text{Mem}(s_2) \ \wedge\ \text{Msg}(\text{H},id,\text{Cache},a,v) \in \text{Mout}(s_2) & \\
\Rightarrow & \text{Cell}(a,v) \in \text{Mem}(\text{dr}(s_1)) \ \wedge\ a \notin \text{Cache}_{id}(\text{dr}(s_1)) \ \wedge & (\textit{Lemmas 5 \& 8}) \\
 & \text{Cell}(a,v) \in \text{Mem}(\text{dr}(s_2)) \ \wedge\ \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s_2)) & (\textit{Lemmas 5 \& 8}) \\
\Rightarrow & \text{f}(s_1) \to \text{f}(s_2) \qquad (\textit{CRF-Cache}) &
\end{array}$$

- If Rule IM2 (*M-Receive-Wb-&-Send-WbAck*) applies, then

$$\begin{array}{lll}
 & \text{Msg}(id,\text{H},\text{Wb},a,v) \in \text{Min}(s_1) \ \wedge & \\
 & \text{Cell}(a,v) \in \text{Mem}(s_2) \ \wedge\ \text{Msg}(\text{H},id,\text{WbAck},a) \in \text{Mout}(s_2) & \\
\Rightarrow & \text{Cell}(a,v,\text{Dirty}) \in \text{Cache}_{id}(\text{dr}(s_1)) \ \wedge & (\textit{Lemma 8}) \\
 & \text{Cell}(a,v) \in \text{Mem}(\text{dr}(s_2)) \ \wedge\ \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s_2)) & (\textit{Lemmas 5 \& 8}) \\
\Rightarrow & \text{f}(s_1) \to \text{f}(s_2) \qquad (\textit{CRF-Writeback}) &
\end{array}$$

Draining Rules

- If Rule IC2 (*C-Send-Wb*), IC3 (*C-Send-CacheReq*), IC4 (*C-Receive-Cache*), IC5 (*C-Receive-WbAck*), *Message-Cache-to-Mem* or *Message-Mem-to-Cache* applies, then

$$\text{f}(s_1) = \text{f}(s_2) \qquad (\textit{Since the rule or its backward version is a draining rule})$$

Figure 4.6 summarizes the simulation proof. □

CRF Rule	**Base Imperative Rules**
CRF-Loadl	*Loadl-on-Clean* or *Loadl-on-Dirty*
CRF-Storel	*Storel-on-Clean* or *Storel-on-Dirty*
CRF-Commit	*Commit-on-Clean* or *Commit-on-Invalid*
CRF-Reconcile	*Reconcile-on-Dirty* or *Reconcile-on-Invalid*
CRF-Cache	*C-Send-CacheReq* + *Message-Cache-to-Mem* + *M-Receive-CacheReq-&-Send-Cache* + *Message-Mem-to-Cache* + *C-Receive-Cache*
CRF-Writeback	*C-Send-Wb* + *Message-Cache-to-Mem* + *M-Receive-Wb-&-Send-WbAck* + *Message-Mem-to-Cache* + *C-Receive-WbAck*
CRF-Purge	*C-Purge*

Figure 4.8: Simulation of CRF in Base

4.5.4 Soundness of Base

The simulation theorem demonstrates that each imperative rule of Base can be simulated by some CRF rules. Figure 4.7 shows that all the Base rules can be derived from the imperative rules (Base has no directive rule). Therefore, the Base protocol is a sound implementation of CRF.

We mention in passing that each CRF rule can also be simulated by a sequence of Base rules. Given a CRF term s, we can lift it to a Base term by adding empty message queues and proper cache identifiers. For example, the cache operation in CRF can be simulated as follows: the cache issues a cache request, the network passes the request to the memory, the memory sends a cache message with the requested data, the network passes the cache message to the cache, and the cache receives the message and caches the data. Figure 4.8 gives the corresponding sequence of Base rules for the simulation of each CRF rule.

4.6 Liveness Proof of the Base Protocol

In this section, we prove the liveness of Base by showing that each processor makes progress, that is, a memory instruction can always be completed eventually. We first present some invariants that will be used in the liveness proof.

4.6.1 Some Invariants of Base

Lemma 11 includes invariants regarding message generation and processing. Invariants (1)-(4) ensure that whenever a Loadl or Storel instruction is performed on an uncached address, the cache will send a CacheReq message to the memory; whenever a Commit instruction is performed on a dirty cell, the cache will send a Wb message to the memory; whenever a Reconcile instruction is performed on a clean cell, the cache will have the address purged. The proof simply follows the weak fairness of Rules P5, P10, P12 and P16.

Invariants (5) and (6) ensure that whenever a cache receives a Cache or WbAck message, it will set the cache state to Clean. Invariants (7) and (8) ensure that whenever the memory receives a CacheReq message, it will send a Cache message to the cache; whenever the memory receives a Wb message, it will send a WbAck message to the cache. These invariants can be proved based on the weak fairness of Rules MC1, MC2, MM1 and MM2.

Invariants (9) and (10) ensure that each outgoing message will be delivered to its destination's incoming queue. This can be proved based on the weak fairness of the message passing rules.

Lemma 11 Given a Base sequence σ,

(1) $\langle t,\text{Loadl}(a)\rangle \in \text{Pmb}_{id}(\sigma) \ \wedge\ a \notin \text{Cache}_{id}(\sigma) \quad \leadsto \quad \text{Msg}(id,\text{H},\text{CacheReq},a) \in \text{Cout}_{id}(\sigma)$

(2) $\langle t,\text{Storel}(a,\text{-})\rangle \in \text{Pmb}_{id}(\sigma) \ \wedge\ a \notin \text{Cache}_{id}(\sigma) \quad \leadsto \quad \text{Msg}(id,\text{H},\text{CacheReq},a) \in \text{Cout}_{id}(\sigma)$

(3) $\langle t,\text{Commit}(a)\rangle \in \text{Pmb}_{id}(\sigma) \ \wedge\ \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma)$
$\qquad\qquad \leadsto \quad \text{Msg}(id,\text{H},\text{Wb},a,\text{-}) \in \text{Cout}_{id}(\sigma)$

(4) $\langle t,\text{Reconcile}(a)\rangle \in \text{Pmb}_{id}(\sigma) \ \wedge\ \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)$
$\qquad\qquad \leadsto \quad a \notin \text{Cache}_{id}(\sigma)$

(5) $\text{Msg}(\text{H},id,\text{Cache},a,\text{-}) \in \text{Cin}_{id}(\sigma) \quad \leadsto \quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)$

(6) $\text{Msg}(\text{H},id,\text{WbAck},a) \in \text{Cin}_{id}(\sigma) \quad \leadsto \quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)$

(7) $\text{Msg}(id,\text{H},\text{CacheReq},a) \in \text{Min}(\sigma) \quad \leadsto \quad \text{Msg}(\text{H},id,\text{Cache},a,\text{-}) \in \text{Mout}(\sigma)$

(8) $\text{Msg}(id,\text{H},\text{Wb},a,\text{-}) \in \text{Min}(\sigma) \quad \leadsto \quad \text{Msg}(\text{H},id,\text{WbAck},a) \in \text{Mout}(\sigma)$

(9) $msg \in \text{Cout}_{id}(\sigma) \ \wedge\ \text{Dest}(msg) = \text{H} \quad \leadsto \quad msg \in \text{Min}(\sigma)$

(10) $msg \in \text{Mout}(\sigma) \ \wedge\ \text{Dest}(msg) = id \quad \leadsto \quad msg \in \text{Cin}_{id}(\sigma)$

Lemma 12 ensures that if an address is cached in the CachePending or WbPending state, it will eventually be cached in the Clean state.

Lemma 12 Given a Base sequence σ,

(1) $\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \quad \leadsto \quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)$

(2) $\text{Cell}(a,\text{-},\text{WbPending}) \in \text{Cache}_{id}(\sigma) \quad \leadsto \quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)$

Proof We first show that if a cache sends a CacheReq or Wb message to the memory, the cache state will become Clean eventually. This can be represented by the following proposition; the proof follows trivially from Lemma 11.

- $\text{Msg}(id,\text{H},\text{CacheReq},a) \in \text{Cout}_{id}(\sigma) \quad \leadsto \quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)$
- $\text{Msg}(id,\text{H},\text{Wb},a,\text{-}) \in \text{Cout}_{id}(\sigma) \quad \leadsto \quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)$

We then show that when a cache changes the state of a cache cell to CachePending or WbPending, it sends a CacheReq or Wb message to the memory. The following proposition can be verified by simply checking each Base rule.

- $\text{Cell}(a,\text{-},\text{CachePending}) \notin \text{Cache}_{id}(\sigma) \ \wedge\ \bigcirc\, \text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma)$
  $\qquad \Rightarrow \quad \bigcirc\, \text{Msg}(id,\text{H},\text{CacheReq},a) \in \text{Cout}_{id}(\sigma)$
- $\text{Cell}(a,\text{-},\text{WbPending}) \notin \text{Cache}_{id}(\sigma) \ \wedge\ \bigcirc\, \text{Cell}(a,\text{-},\text{WbPending}) \in \text{Cache}_{id}(\sigma)$
  $\qquad \Rightarrow \quad \bigcirc\, \text{Msg}(id,\text{H},\text{Wb},a,\text{-}) \in \text{Cout}_{id}(\sigma)$

This completes the proof according to Theorem-A (see Section 2.5). □

4.6.2 Liveness of Base

Lemma 13 ensures that whenever a processor intends to execute an instruction, the cache cell will be set to an appropriate state while the instruction remains in the processor-to-memory buffer. For a Loadl or Storel, the cache state will be set to Clean or Dirty; for a Commit, the cache state will be set to Clean or Invalid; for a Reconcile, the cache state will be set to Dirty or Invalid.

Lemma 13 Given a Base sequence σ,

(1) $\text{Loadl}(a) \in \text{Pmb}_{id}(\sigma) \;\leadsto\; \text{Loadl}(a) \in \text{Pmb}_{id}(\sigma) \;\wedge$
$\quad(\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma))$

(2) $\text{Storel}(a,\text{-}) \in \text{Pmb}_{id}(\sigma) \;\leadsto\; \text{Storel}(a,\text{-}) \in \text{Pmb}_{id}(\sigma) \;\wedge$
$\quad(\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma))$

(3) $\text{Commit}(a) \in \text{Pmb}_{id}(\sigma) \;\leadsto\; \text{Commit}(a) \in \text{Pmb}_{id}(\sigma) \;\wedge$
$\quad(\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee\; a \notin \text{Cache}_{id}(\sigma))$

(4) $\text{Reconcile}(a) \in \text{Pmb}_{id}(\sigma) \;\leadsto\; \text{Reconcile}(a) \in \text{Pmb}_{id}(\sigma) \;\wedge$
$\quad(\text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \;\vee\; a \notin \text{Cache}_{id}(\sigma))$

Proof We first show that when a processor intends to execute an instruction, the cache cell will be brought to an appropriate state. This can be represented by the following proposition; the proof follows from Lemmas 11 and 12.

- $\text{Loadl}(a) \in \text{Pmb}_{id}(\sigma) \;\leadsto\; \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma)$
- $\text{Storel}(a,\text{-}) \in \text{Pmb}_{id}(\sigma) \;\leadsto\; \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma)$
- $\text{Commit}(a) \in \text{Pmb}_{id}(\sigma) \;\leadsto\; \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee\; a \notin \text{Cache}_{id}(\sigma)$
- $\text{Reconcile}(a) \in \text{Pmb}_{id}(\sigma) \;\leadsto\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \;\vee\; a \notin \text{Cache}_{id}(\sigma)$

We then show that an instruction can be completed only when the address is cached in an appropriate state. This can be represented by the following proposition, which can be verified by simply checking each Base rule.

- $\langle t,\text{Loadl}(a)\rangle \in \text{Pmb}_{id}(\sigma) \;\wedge\; \circ\,\langle t,\text{Loadl}(a)\rangle \notin \text{Pmb}_{id}(\sigma)$
  $\quad\Rightarrow\; \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma)$
- $\langle t,\text{Storel}(a,\text{-})\rangle \in \text{Pmb}_{id}(\sigma) \;\wedge\; \circ\,\langle t,\text{Storel}(a,\text{-})\rangle \notin \text{Pmb}_{id}(\sigma)$
  $\quad\Rightarrow\; \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma)$
- $\langle t,\text{Commit}(a)\rangle \in \text{Pmb}_{id}(\sigma) \;\wedge\; \circ\,\langle t,\text{Commit}(a)\rangle \notin \text{Pmb}_{id}(\sigma)$
  $\quad\Rightarrow\; \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee\; a \notin \text{Cache}_{id}(\sigma)$
- $\langle t,\text{Reconcile}(a)\rangle \in \text{Pmb}_{id}(\sigma) \;\wedge\; \circ\,\langle t,\text{Reconcile}(a)\rangle \notin \text{Pmb}_{id}(\sigma)$
  $\quad\Rightarrow\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \;\vee\; a \notin \text{Cache}_{id}(\sigma)$

This completes the proof according to Theorem-B (see Section 2.5). □

Lemma 13 ensures that when a processor intends to execute an instruction, the cache cell will be brought into an appropriate state so that the instruction can be completed. However, this does not guarantee that the instruction will be completed since a cache state can change at any time because of voluntary rules. To ensure that each processor makes progress, an instruction must be completed if it has an infinite number of opportunities to be executed. This is guaranteed by the strong fairness of Rules P1, P2, P6, P7, P11, P15, P17 and P20.

Theorem 14 (Liveness of Base) Given a Base sequence σ,

(1) $\langle t,\text{Loadl(-)}\rangle \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \langle t,\text{-}\rangle \in \text{Mpb}_{id}(\sigma)$
(2) $\langle t,\text{Storel(-,-)}\rangle \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \langle t,\text{Ack}\rangle \in \text{Mpb}_{id}(\sigma)$
(3) $\langle t,\text{Commit(-)}\rangle \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \langle t,\text{Ack}\rangle \in \text{Mpb}_{id}(\sigma)$
(4) $\langle t,\text{Reconcile(-)}\rangle \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \langle t,\text{Ack}\rangle \in \text{Mpb}_{id}(\sigma)$

The liveness proof assumes that there can be at most one outstanding memory instruction in each processor-to-memory buffer. It is obvious that the proof still holds in the presence of multiple outstanding memory instructions, provided that the reordering mechanism ensures fair scheduling of outstanding instructions. An implementation can enforce such fairness by requiring that a stalled instruction be retried repeatedly until it is retired.

Chapter 5

The Writer-Push Cache Coherence Protocol

In Base, a reconcile operation on a clean cell forces the data to be purged from the cache before the reconcile can complete. This may cause serious performance degradation for programs with excessive use of reconcile operations. For example, this can happen for programs written under release consistency that requires all addresses be indistinguishably reconciled after each acquire operation. The Writer-Push (WP) protocol allows a reconcile operation on a clean cell to complete without purging the data so that the data can be accessed by subsequent instructions without causing a cache miss. It is ideal for programs with excessive use of reconcile operations, for example, when some processors read a memory location many times using Reconcile and Loadl instructions before the location is modified by another processor using Storel and Commit instructions. A cache cell never needs to be purged from the cache unless the cache becomes full or the memory location is modified by another processor.

The WP protocol ensures that if an address is cached in the Clean state, the cache cell contains the same value as the memory cell. This is achieved by requiring that all clean copies of an address be purged before the memory cell can be modified. As the name "Writer-Push" suggests, the writer is responsible for informing potential readers to have their stale copies, if any, purged in time. Therefore, a commit operation on a dirty cell can be a lengthy process since it cannot complete before clean copies of the address are purged from all other caches.

Section 5.1 describes the system configuration of the WP protocol, which includes the coherence states and protocol messages. We present the imperative rules of WP in Section 5.2 and the integrated rules of WP in Section 5.3. The soundness and liveness of WP are proved in Sections 5.4 and 5.5, respectively. Section 5.6 demonstrates the use of voluntary rules via an update protocol that is derived from WP. An alternative Writer-Push protocol is given in Section 5.7.

SYS	$\equiv$	Sys(MSITE, SITEs)	*System*
MSITE	$\equiv$	Msite(MEM, IN, OUT)	*Memory Site*
MEM	$\equiv$	ϵ [] Cell(a,v,MSTATE) $\mid$ MEM	*Memory*
SITEs	$\equiv$	SITE [] SITE $\mid$ SITEs	*Set of Cache Sites*
SITE	$\equiv$	Site(id, CACHE, IN, OUT, PMB, MPB, PROC)	*Cache Site*
CACHE	$\equiv$	ϵ [] Cell(a,v,CSTATE) $\mid$ CACHE	*Cache*
IN	$\equiv$	ϵ [] MSG $\odot$ IN	*Incoming Queue*
OUT	$\equiv$	ϵ [] MSG $\otimes$ OUT	*Outgoing Queue*
MSG	$\equiv$	Msg(src,$dest$,CMD,a,v)	*Message*
MSTATE	$\equiv$	C[DIR] [] T[DIR,SM]	*Memory state*
DIR	$\equiv$	ϵ [] id $\mid$ DIR	*Directory*
SM	$\equiv$	ϵ [] (id,v) $\mid$ SM	*Suspended Messages*
CSTATE	$\equiv$	Clean [] Dirty [] WbPending [] CachePending	*Cache State*
CMD	$\equiv$	Cache [] Purge [] Wb [] WbAck []	
		FlushAck [] CacheReq [] PurgeReq	*Command*

Figure 5.1: System Configuration of WP

5.1 The System Configuration of the WP Protocol

Figure 5.1 defines the system configuration for the WP protocol. The system contains a memory site and a set of cache sites. The memory site has three components, a memory, an incoming queue and an outgoing queue. Each cache site contains an identifier, a cache, an incoming queue, an outgoing queue, a processor-to-memory buffer, a memory-to-processor buffer and a processor. Different cache sites have different cache identifiers.

The WP protocol employs two stable cache states, Clean and Dirty, and two transient cache states, WbPending and CachePending. The WbPending state means a writeback operation is being performed on the address, and the CachePending state means a cache copy is being requested for the address. Each memory cell maintains a memory state, which can be C[*dir*] or T[*dir*,*sm*], where *dir* contains identifiers of the cache sites in which the address is cached, and *sm* contains suspended writeback messages (only the source and the data are recorded). In terms of soundness, CachePending is identical to Invalid, and C[*dir*] is identical to T[*dir*,ϵ]. The CachePending and C[*dir*] states are introduced purely for liveness reason, and are not used in imperative rules.

Figure 5.2 shows the imperative and directive messages of WP. Informally, the meaning of each protocol message is as follows:

- Cache: the memory supplies a data copy to the cache.
- WbAck: the memory acknowledges a writeback operation and allows the cache to retain a clean copy.
- FlushAck: the memory acknowledges a writeback operation and requires the cache to purge the address.
- PurgeReq: the memory requests the cache to purge a clean copy.

	From Memory to Cache	**From Cache to Memory**
Imperative Messages	$\mathrm{Msg}(\mathrm{H},id,\mathrm{Cache},a,v)$ $\mathrm{Msg}(\mathrm{H},id,\mathrm{WbAck},a)$ $\mathrm{Msg}(\mathrm{H},id,\mathrm{FlushAck},a)$	$\mathrm{Msg}(id,\mathrm{H},\mathrm{Purge},a)$ $\mathrm{Msg}(id,\mathrm{H},\mathrm{Wb},a,v)$
Directive Messages	$\mathrm{Msg}(\mathrm{H},id,\mathrm{PurgeReq},a)$	$\mathrm{Msg}(id,\mathrm{H},\mathrm{CacheReq},a)$

Figure 5.2: Protocol Messages of WP

- Purge: the cache informs the memory of a purge operation.
- Wb: the cache writes a dirty copy back to the memory.
- CacheReq: the cache requests a data copy from the memory.

5.2 The Imperative Rules of the WP Protocol

We develop a set of imperative rules that specify all coherence actions that determine the soundness of the system. This includes three sets of rules, the processor rules, the cache engine rules and the memory engine rules.

Processor Rules: The imperative processor rules of WP contain all the imperative processor rules of Base (see Section 4.3), and the *Reconcile-on-Clean* rule that allows a Reconcile instruction to complete even when the address is cached in the Clean state.

Reconcile-on-Clean Rule

$\quad\mathrm{Site}(id,\ \mathrm{Cell}(a,v,\mathrm{Clean})\mid cache,\ in,\ out,\ \langle t,\mathrm{Reconcile}(a)\rangle;pmb,\ mpb,\ proc)$

$\rightarrow\ \mathrm{Site}(id,\ \mathrm{Cell}(a,v,\mathrm{Clean})\mid cache,\ in,\ out,\ pmb,\ mpb|\langle t,\mathrm{Ack}\rangle,\ proc)$

C-engine Rules: A cache can purge a clean cell and inform the memory via a Purge message. It can also write the data of a dirty cell to the memory via a Wb message and set the cache state to WbPending, indicating that a writeback operation is being performed on the address. There are two possible acknowledgments for a writeback operation. If a writeback acknowledgment (WbAck) is received, the cache state becomes Clean; if a flush acknowledgment (FlushAck) is received, the cache state becomes Invalid (that is, the address is purged from the cache).

When a cache receives a Cache message, it simply caches the data in the Clean state. Unlike Base, a cache can receive a data copy from the memory even though it has not requested for the data. Figure 5.3 shows cache state transitions due to imperative operations.

C-Send-Purge Rule

$\quad\mathrm{Site}(id,\ \mathrm{Cell}(a,\text{-},\mathrm{Clean})\mid cache,\ in,\ out,\ pmb,\ mpb,\ proc)$

$\rightarrow\ \mathrm{Site}(id,\ cache,\ in,\ out\otimes\mathrm{Msg}(id,\mathrm{H},\mathrm{Purge},a),\ pmb,\ mpb,\ proc)$

C-Send-Wb Rule

$\quad\mathrm{Site}(id,\ \mathrm{Cell}(a,v,\mathrm{Dirty})\mid cache,\ in,\ out,\ pmb,\ mpb,\ proc)$

$\rightarrow\ \mathrm{Site}(id,\ \mathrm{Cell}(a,v,\mathrm{WbPending})\mid cache,\ in,\ out\otimes\mathrm{Msg}(id,\mathrm{H},\mathrm{Wb},a,v),\ pmb,\ mpb,\ proc)$

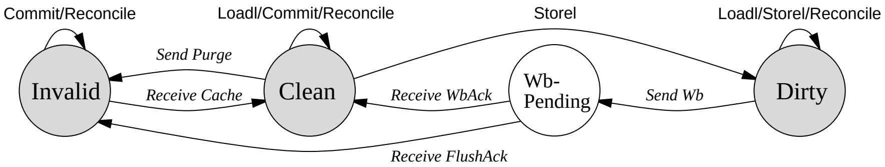


Figure 5.3: Cache State Transitions of WP's Imperative Operations

C-Receive-WbAck Rule

$\quad$ Site(id, Cell(a,v,WbPending) | $cache$, Msg(H,id,WbAck,a) $\odot$ in, out, pmb, mpb, $proc$)

$\rightarrow$ Site(id, Cell(a,v,Clean) | $cache$, in, out, pmb, mpb, $proc$)

C-Receive-FlushAck Rule

$\quad$ Site(id, Cell(a,-,WbPending) | $cache$, Msg(H,id,FlushAck,a) $\odot$ in, out, pmb, mpb, $proc$)

$\rightarrow$ Site(id, $cache$, in, out, pmb, mpb, $proc$)

C-Receive-Cache Rule

$\quad$ Site(id, $cache$, Msg(H,id,Cache,a,v) $\odot$ in, out, pmb, mpb, $proc$)

$\rightarrow$ Site(id, Cell(a,v,Clean) | $cache$, in, out, pmb, mpb, $proc$)

M-engine Rules: The memory can send a data copy of an address to a cache in which the directory shows the address is not cached; the directory is updated accordingly. When the memory receives a purge message, it simply deletes the cache identifier from the directory.

When the memory receives a writeback message, it suspends the message and removes the corresponding cache identifier from the directory. A suspended writeback message can be resumed if the directory shows that the address is not cached in any cache. In this case, the memory updates its value and sends a flush acknowledgment back to the cache site. If the writeback message is the only suspended message, the memory can send a writeback acknowledgment instead of a flush acknowledgment to allow the cache to retain a clean copy.

M-Send-Cache Rule

$\quad$ Msite(Cell(a,v,T[dir,ϵ]) | mem, in, out) $\quad$ *if* $\quad id \notin dir$

$\rightarrow$ Msite(Cell(a,v,T[id|dir,ϵ]) | mem, in, out $\otimes$ Msg(H,id,Cache,a,v))

M-Receive-Purge Rule

$\quad$ Msite(Cell(a,v,T[id|dir,sm]) | mem, Msg(id,H,Purge,a) $\odot$ in, out)

$\rightarrow$ Msite(Cell(a,v,T[dir,sm]) | mem, in, out)

M-Receive-Wb Rule

$\quad$ Msite(Cell(a,v_1,T[id|dir,sm]) | mem, Msg(id,H,Wb,a,v) $\odot$ in, out)

$\rightarrow$ Msite(Cell(a,v_1,T[dir,sm|(id,v)]) | mem, in, out)

M-Send-FlushAck Rule

$\quad$ Msite(Cell(a,-,T[ϵ,(id,v)|sm]) | mem, in, out)

$\rightarrow$ Msite(Cell(a,v,T[ϵ,sm]) | mem, in, out $\otimes$ Msg(H,id,FlushAck,a))

Imperative Processor Rules				
Instruction	**Cstate**	**Action**	**Next Cstate**	
Loadl(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	IP1
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	IP2
Storel(a,v)	Cell(a,-,Clean)	*retire*	Cell(a,v,Dirty)	IP3
	Cell(a,-,Dirty)	*retire*	Cell(a,v,Dirty)	IP4
Commit(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	IP5
	$a \notin cache$	*retire*	$a \notin cache$	IP6
Reconcile(a)	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	IP7
	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	IP8
	$a \notin cache$	*retire*	$a \notin cache$	IP9

Imperative C-engine Rules				
Msg from H	**Cstate**	**Action**	**Next Cstate**	
	Cell(a,-,Clean)	$\langle$Purge,$a\rangle \rightarrow$ H	$a \notin cache$	IC1
	Cell(a,v,Dirty)	$\langle$Wb,a,$v\rangle \rightarrow$ H	Cell(a,v,WbPending)	IC2
$\langle$WbAck,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean)	IC3
$\langle$FlushAck,$a\rangle$	Cell(a,-,WbPending)		$a \notin cache$	IC4
$\langle$Cache,a,$v\rangle$	$a \notin cache$		Cell(a,v,Clean)	IC5

Imperative M-engine Rules				
Msg from id	**Mstate**	**Action**	**Next Mstate**	
	Cell(a,v,T[dir,ϵ]) ($id \notin dir$)	$\langle$Cache,a,$v\rangle \rightarrow id$	Cell(a,v,T[$id \mid dir$,ϵ])	IM1
$\langle$Purge,$a\rangle$	Cell(a,v,T[$id \mid dir$,sm])		Cell(a,v,T[dir,sm])	IM2
$\langle$Wb,a,$v\rangle$	Cell(a,v_1,T[$id \mid dir$,sm])		Cell(a,v_1,T[dir,$sm \mid (id,v)$])	IM3
	Cell(a,-,T[ϵ,$(id,v) \mid sm$])	$\langle$FlushAck,$a\rangle \rightarrow id$	Cell(a,v,T[ϵ,sm])	IM4
	Cell(a,-,T[ϵ,(id,v)])	$\langle$WbAck,$a\rangle \rightarrow id$	Cell(a,v,T[id,ϵ])	IM5

Figure 5.4: Imperative Rules of WP

M-Send-WbAck Rule

$$\begin{aligned} & \text{Msite}(\text{Cell}(a,\text{-},\text{T}[\epsilon,(id,v)]) \mid mem,\ in,\ out) \\ \rightarrow\ & \text{Msite}(\text{Cell}(a,v,\text{T}[id,\epsilon]) \mid mem,\ in,\ out \otimes \text{Msg}(\text{H},id,\text{WbAck},a)) \end{aligned}$$

Note that when a writeback message is suspended, the imperative rules rely on an oracle to inform the caches to purge or write back their copies so that the suspended message can be eventually resumed. The Imperative-&-Directive design methodology allows us to separate the soundness and liveness concerns.

Figure 5.4 summarizes the imperative rules of WP. When an instruction is retired, it is removed from the processor-to-memory buffer while an appropriate response is supplied to the corresponding memory-to-processor buffer. When a message is received, it is removed from the incoming queue.

5.3 The WP Protocol

The imperative rules define all the coherence actions that can affect the soundness of the system; they intentionally do not address the liveness issue. For example, when the memory receives a writeback message, it suspends the message in the memory state. A suspended message can be resumed when the directory shows that the address is no longer cached in any cache. However, the imperative rules do not specify how to bring the system to such a state that a suspended message can be resumed eventually. To ensure liveness, the memory must send a purge request to the caches in which the address is cached to force the address to be purged.

We introduce two directive messages, CacheReq and PurgeReq, for this purpose. A cache can send a CacheReq message to the memory to request for a data copy; the memory can send a PurgeReq message to a cache to force a clean copy to be purged or a dirty copy to be written back. In addition, we maintain some information about outstanding directive messages by splitting certain imperative cache and memory states. The Invalid state in the imperative rules corresponds to Invalid and CachePending in the integrated protocol, where CachePending implies that a CacheReq message has been sent to the memory. The T[*dir*,ϵ] state in the imperative rules corresponds to T[*dir*,ϵ] and C[*dir*] in the integrated protocol, where T[*dir*,ϵ] implies that a PurgeReq message has been multicast to cache sites *dir*.

Figure 5.5 defines the rules of the WP protocol. The tabular description can be easily translated into formal TRS rules (cases that are not specified represent illegal or unreachable states). The processor rules are all mandatory rules. The cache engine and memory engine rules are categorized into mandatory and voluntary rules. Each mandatory rule is weakly fair in that if it can be applied, it must be applied eventually or become impossible to apply at some time. A mandatory rule marked with 'SF' means the rule requires strong fairness to ensure the liveness of the system. The notation '$msg \rightarrow dir$' means sending the message *msg* to the destinations represented by directory *dir*.

A processor rule may retire or stall an instruction depending on the cache state of the address. When an instruction is retired, it is removed from the processor-to-memory buffer and the corresponding value or reply is sent to the memory-to-processor buffer. When an instruction is stalled, it remains in the processor-to-memory buffer for later processing. A stalled instruction does not necessarily block other instructions to be executed.

An incoming message can be processed or stalled depending on the memory state of the address. When a message is processed, it is removed from the incoming queue. When a message is stalled, it remains in the incoming queue for later processing (only CacheReq messages can be stalled). A stalled message is buffered properly to avoid blocking following messages.

5.3.1 Mandatory Rules

The processor rules are similar to those in the Base protocol, except that a Reconcile instruction can complete even when the address is cached in the Clean state. On a cache miss, the cache sends a cache request to the memory and sets the cache state to be CachePending until the requested data is received. On a Commit instruction, if the address is cached in the Dirty state, the cache writes the data back to the memory and sets the cache state to be WbPending until a writeback acknowledgment is received. An instruction remains stalled when it cannot be completed in the current cache state.

When the memory receives a CacheReq message, it processes the message according to the memory state. If the address is already cached in the cache, the cache request is discarded.

Mandatory Processor Rules

Instruction	**Cstate**	**Action**	**Next Cstate**		
Loadl(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	P1	SF
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	P2	SF
	Cell(a,v,WbPending)	*stall*	Cell(a,v,WbPending)	P3	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P4	
	$a \notin cache$	*stall* $\langle \text{CacheReq}, a \rangle \rightarrow \text{H}$	Cell(a,-,CachePending)	P5	
Storel(a,v)	Cell(a,-,Clean)	*retire*	Cell(a,v,Dirty)	P6	SF
	Cell(a,-,Dirty)	*retire*	Cell(a,v,Dirty)	P7	SF
	Cell(a,v_1,WbPending)	*stall*	Cell(a,v_1,WbPending)	P8	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P9	
	$a \notin cache$	*stall* $\langle \text{CacheReq}, a \rangle \rightarrow \text{H}$	Cell(a,-,CachePending)	P10	
Commit(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	P11	SF
	Cell(a,v,Dirty)	*stall* $\langle \text{Wb}, a, v \rangle \rightarrow \text{H}$	Cell(a,v,WbPending)	P12	
	Cell(a,v,WbPending)	*stall*	Cell(a,v,WbPending)	P13	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P14	
	$a \notin cache$	*retire*	$a \notin cache$	P15	SF
Reconcile(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	P16	SF
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	P17	SF
	Cell(a,v,WbPending)	*stall*	Cell(a,v,WbPending)	P18	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P19	
	$a \notin cache$	*retire*	$a \notin cache$	P20	SF

Voluntary C-engine Rules

	Cstate	**Action**	**Next Cstate**	
	Cell(a,-,Clean)	$\langle \text{Purge}, a \rangle \rightarrow \text{H}$	$a \notin cache$	VC1
	Cell(a,v,Dirty)	$\langle \text{Wb}, a, v \rangle \rightarrow \text{H}$	Cell(a,v,WbPending)	VC2
	$a \notin cache$	$\langle \text{CacheReq}, a \rangle \rightarrow \text{H}$	Cell(a,-,CachePending)	VC3

Mandatory C-engine Rules

Msg from H	**Cstate**	**Action**	**Next Cstate**	
$\langle \text{Cache}, a, v \rangle$	$a \notin cache$		Cell(a,v,Clean)	MC1
	Cell(a,-,CachePending)		Cell(a,v,Clean)	MC2
$\langle \text{WbAck}, a \rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean)	MC3
$\langle \text{FlushAck}, a \rangle$	Cell(a,-,WbPending)		$a \notin cache$	MC4
$\langle \text{PurgeReq}, a \rangle$	Cell(a,-,Clean)	$\langle \text{Purge}, a \rangle \rightarrow \text{H}$	$a \notin cache$	MC5
	Cell(a,v,Dirty)	$\langle \text{Wb}, a, v \rangle \rightarrow \text{H}$	Cell(a,v,WbPending)	MC6
	Cell(a,v,WbPending)		Cell(a,v,WbPending)	MC7
	Cell(a,-,CachePending)		Cell(a,-,CachePending)	MC8
	$a \notin cache$		$a \notin cache$	MC9

Voluntary M-engine Rules

	Mstate	**Action**	**Next Mstate**	
	Cell(a,v,C[dir]) ($id \notin dir$)	$\langle \text{Cache}, a, v \rangle \rightarrow id$	Cell(a,v,C[$id \mid dir$])	VM1
	Cell(a,v,C[dir]) ($dir \neq \epsilon$)	$\langle \text{PurgeReq}, a \rangle \rightarrow dir$	Cell(a,v,T[dir,ϵ])	VM2

Mandatory M-engine Rules

Msg from *id*	**Mstate**	**Action**	**Next Mstate**		
$\langle \text{CacheReq}, a \rangle$	Cell(a,v,C[dir]) ($id \notin dir$)	$\langle \text{Cache}, a, v \rangle \rightarrow id$	Cell(a,v,C[$id \mid dir$])	MM1	SF
	Cell(a,v,C[dir]) ($id \in dir$)		Cell(a,v,C[dir])	MM2	
	Cell(a,v,T[dir,sm]) ($id \notin dir$)	*stall message*	Cell(a,v,T[dir,sm])	MM3	
	Cell(a,v,T[dir,sm]) ($id \in dir$)		Cell(a,v,T[dir,sm])	MM4	
$\langle \text{Wb}, a, v \rangle$	Cell(a,v_1,C[$id \mid dir$])	$\langle \text{PurgeReq}, a \rangle \rightarrow dir$	Cell(a,v_1,T[dir,(id,v)])	MM5	
	Cell(a,v_1,T[$id \mid dir$,sm])		Cell(a,v_1,T[dir,(id,v)$\mid sm$])	MM6	
$\langle \text{Purge}, a \rangle$	Cell(a,v,C[$id \mid dir$])		Cell(a,v,C[dir])	MM7	
	Cell(a,v,T[$id \mid dir$,sm])		Cell(a,v,T[dir,sm])	MM8	
	Cell(a,-,T[ϵ,(id,v)$\mid sm$])	$\langle \text{FlushAck}, a \rangle \rightarrow id$	Cell(a,v,T[ϵ,sm])	MM9	
	Cell(a,-,T[ϵ,(id,v)])	$\langle \text{WbAck}, a \rangle \rightarrow id$	Cell(a,v,C[id])	MM10	
	Cell(a,v,T[ϵ,ϵ])		Cell(a,v,C[ϵ])	MM11	

Figure 5.5: The WP Protocol

If the memory state shows that the address is uncached in the cache, there are two possible cases. The cache request is stalled for later processing if the memory state is a transient state; otherwise the memory sends a cache message to supply the data to the requesting cache.

When a cache request is stalled, it is buffered properly so that following messages in the incoming queue can still be processed. This can be modeled by allowing incoming messages to be reordered with each other under certain conditions. For the WP protocol, the minimal requirement of buffer management is that a stalled message cannot block any following message that has a different source or different address. This requirement can be described as follows:

WP's buffer management:

$$\begin{aligned} & \mathit{msg}_1 \odot \mathit{msg}_2 \quad \equiv \quad \mathit{msg}_2 \odot \mathit{msg}_1 \\ & \qquad \textit{if} \quad (\mathrm{Cmd}(\mathit{msg}_1) = \mathrm{CacheReq} \ \vee\ \mathrm{Cmd}(\mathit{msg}_2) = \mathrm{CacheReq}) \ \wedge \\ & \qquad\qquad (\mathrm{Src}(\mathit{msg}_1) \neq \mathrm{Src}(\mathit{msg}_2) \ \vee\ \mathrm{Addr}(\mathit{msg}_1) \neq \mathrm{Addr}(\mathit{msg}_2)) \end{aligned}$$

There can be several stalled requests regarding the same address from different cache sites. The strong fairness of Rule MM1 prevents a cache request from being stalled forever while cache requests or writeback messages from other caches are serviced again and again. In practice, we can use a FIFO queue for buffered messages to ensure such fairness.

The role of the memory is more complicated for writeback operations, because the memory must ensure that other cache copies of the same address are coherent. This can be achieved by multicasting a purge request to all the caches recorded in the directory, except the one from which the writeback is received. The writeback acknowledgment is withheld until the memory has received acknowledgements for all the purge requests. The transient state T[*dir*,*sm*] is used for the bookkeeping purpose in the memory. In the transient state, *dir* represents the cache sites which have not yet acknowledged the purge requests, and *sm* contains the suspended writeback message that the memory has received but has not yet acknowledged (only the source and the data need to be recorded).

Each time the memory receives a purge acknowledgment, the directory is updated accordingly. The suspended writeback message is resumed after the directory becomes empty, that is, all the purge requests have been acknowledged. The memory can then update the value of the memory cell and send a writeback acknowledgment (WbAck) or a flush acknowledgment (FlushAck) to the cache. If the cache receives a WbAck acknowledgment, it retains a clean copy; otherwise it purges its copy.

If the memory receives more than one writeback message, it records all the writeback messages in the transient state. The suspended messages are resumed when the directory becomes empty. The memory acknowledges each writeback message via a FlushAck message (it may chose to acknowledge the last writeback message via a WbAck message since the cache contains the same value as the memory). This ensures that all the stale copies of the address are purged from the caches.

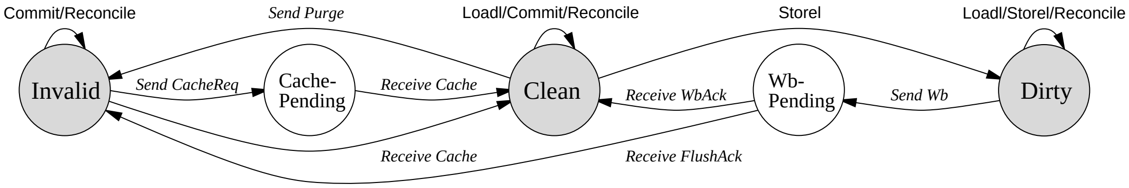


Figure 5.6: Cache State Transitions of WP

On the other hand, a cache responds to a purge request on a clean cell by purging the clean data and sending a Purge message to the memory. In the case that the cache copy is dirty, the dirty copy is forced to be written back via a Wb message.

5.3.2 Voluntary Rules

At any time, a cache can purge a clean cell, and notify the memory of the purge operation via a Purge message. It can also write a dirty copy back to the memory via a Wb message. Furthermore, a cache can send a message to the memory to request a data copy for any uncached address, even though no Loadl or Storel instruction is performed. Figure 5.6 shows the cache state transitions of WP.

The memory can voluntarily send a data copy to any cache, provided the directory shows that the address is not cached in that cache. This implies that a cache may receive a data copy even though it has not requested for it. The memory can also voluntarily multicast a purge request to purge clean copies of an address.

The voluntary rules allow the memory to supply a data copy without a request from the cache, and a cache to purge or write back a data copy without a request from the memory. This can cause unexpected situations if a request is received after the requested action has been performed. For example,

- Simultaneous Cache and CacheReq. Suppose initially the address is not cached in a cache site. The memory sends a Cache message to the cache, while the cache sends a CacheReq message to the memory. The CacheReq message will be discarded when it is received at the memory (Rules MM2 & MM4).
- Simultaneous Purge and PurgeReq. Suppose initially a clean copy of the address is cached in a cache site. The cache purges the clean copy and sends a Purge message to the memory, while the memory sends a PurgeReq message to the cache. The PurgeReq message will be discarded when it is received at the cache (Rules MC8 & MC9).
- Simultaneous Wb and PurgeReq. Suppose initially a dirty copy of the address is cached in a cache site. The cache sends a Wb message to write the dirty copy back to the memory, while the memory sends a PurgeReq message to the cache. The PurgeReq message will be discarded when it is received at the cache (Rule MC7).

5.3.3 FIFO Message Passing

The liveness of WP is contingent upon FIFO message passing. Consider a scenario in which the memory sends a Cache message to supply a data copy to a cache, followed by a PurgeReq message to request the cache copy to be purged or written back. According to Rules MC8 and MC9, the PurgeReq message would be discarded if it were received before the Cache message, which would inevitably lead to a deadlock or livelock situation. The minimal requirement is that FIFO ordering must be maintained for the following messages:

- FlushAck followed by Cache. The memory sends a FlushAck message to acknowledge a writeback operation, and then sends a Cache message to supply a data copy to the cache.
- Cache followed by PurgeReq. The memory sends a Cache message to a cache to supply a data copy, and then sends a PurgeReq message to purge the data copy.
- WbAck followed by PurgeReq. The memory sends a WbAck message to acknowledge a writeback operation (the cache is allowed to retain a clean copy), and then sends a PurgeReq message to purge the data copy.
- Purge followed by CacheReq. A cache sends a Purge message to the memory after purging a clean copy, and then sends a CacheReq message to the memory to request for a data copy.

It is worth mentioning that FIFO message passing is not always necessary when the preceding message is a directive message. For example, if a cache sends a CacheReq message followed by a Purge message, the two messages can be received out-of-order without incurring deadlock or livelock. In this case, the CacheReq message, which would be discarded under FIFO message passing, invokes the memory to send a data copy to the cache. Therefore, directive messages can be treated as low-priority messages in that they can be overtaken by imperative messages. The WP protocol does not require FIFO ordering in the following scenarios:

- CacheReq followed by Purge. A cache sends a CacheReq message to the memory. Before the CacheReq message is received, the memory voluntarily sends a Cache message to the cache. The cache receives the Cache message and caches the data in the Clean state. The cache then purges the clean copy and sends a Purge message to the memory. The CacheReq message and the Purge message can be reordered.
- CacheReq followed by Wb. A cache sends a CacheReq message to the memory. Before the CacheReq message is received, the memory voluntarily sends a Cache message to the cache. The cache receives the Cache message and caches the data in the Clean state. The processor performs a Storel instruction, and the cache state becomes Dirty. The cache then sends a Wb message to write the dirty copy back to the memory. The CacheReq message and the Wb message can be reordered.
- PurgeReq followed by Cache. The memory sends a PurgeReq message to a cache, while the address is cached in the Clean state in the cache. Before the PurgeReq message is

Voluntary M-engine Rules			
	Mstate	**Action**	**Next Mstate**
	Cell(a,v,C[dir]) ($id \notin dir$)	$\langle$Cache,a,$v\rangle \rightarrow id$	Cell(a,v,C[$id \mid dir$])
	Cell(a,v,C[dir]) ($dir \neq \epsilon$)	$\langle$PurgeReq,$a\rangle \rightarrow dir$	Cell(a,v,T[dir,ϵ])

Mandatory M-engine Rules				
Msg from id	**Mstate**	**Action**	**Next Mstate**	
$\langle$CacheReq,$a\rangle$	Cell(a,v,C[dir]) ($id \notin dir$)	$\langle$Cache,a,$v\rangle \rightarrow id$	Cell(a,v,C[$id \mid dir$])	SF
	Cell(a,v,C[dir]) ($id \in dir$)		Cell(a,v,C[dir])	
	Cell(a,v,T[dir,dir_1]) ($id \notin dir$)	*stall message*	Cell(a,v,T[dir,dir_1])	
	Cell(a,v,T[dir,dir_1]) ($id \in dir$)		Cell(a,v,T[dir,dir_1])	
$\langle$Wb,a,$v\rangle$	Cell(a,-,C[$id \mid dir$])	$\langle$PurgeReq,$a\rangle \rightarrow dir$	Cell(a,v,T[dir,id])	
	Cell(a,-,T[$id \mid dir$,ϵ])		Cell(a,v,T[dir,id])	
	Cell(a,v_1,T[$id \mid dir$,dir_1]) ($dir_1 \neq \epsilon$)		Cell(a,v_1,T[dir,$id \mid dir_1$])	
$\langle$Purge,$a\rangle$	Cell(a,v,C[$id \mid dir$])		Cell(a,v,C[dir])	
	Cell(a,v,T[$id \mid dir$,dir_1])		Cell(a,v,T[dir,dir_1])	
	Cell(a,v,T[ϵ,$id \mid dir_1$])	$\langle$FlushAck,$a\rangle \rightarrow id$	Cell(a,v,T[ϵ,dir_1])	
	Cell(a,v,T[ϵ,ϵ])		Cell(a,v,C[ϵ])	

Figure 5.7: Simplified Memory Engine Rules of WP

received, the cache voluntarily purges the clean copy and sends a Purge message to the memory. The memory receives the Purge message, and then sends a Cache message to the cache. The PurgeReq message and the Cache message can be reordered.

- PurgeReq followed by WbAck or FlushAck. The memory sends a PurgeReq message to a cache, while the address is cached in the Dirty state in the cache. Before the PurgeReq message is received, the cache voluntarily sends a Wb message to write the data back to the memory. The memory receives the Wb message, and then sends a WbAck or FlushAck message to the cache to acknowledge the writeback operation. The PurgeReq message and the WbAck or FlushAck message can be reordered.

5.3.4 Potential Optimizations

In WP, an instruction is stalled when the address is cached in a transient state. This constraint can be relaxed under certain circumstances. For example, a Commit or Reconcile instruction can be completed when the address is cached in the CachePending or WbPending state. This optimization is useful since a cache may voluntarily request for a data copy from the memory or voluntarily write a dirty copy back to the memory. It is desirable that such voluntary actions do not block instruction execution unnecessarily.

The WP protocol requires that, when a writeback message is suspended, both the source and the data be recorded in the transient memory state. This preserves the original data of the memory cell which is needed for backward draining in the soundness proof. In practice, the data of a writeback message can be directly written to the memory cell. Furthermore, when there are several writeback messages from different cache sites, the memory is updated only once. Since the value of any writeback message can be used to update the memory, we simply use the value of the first writeback message and discard the values of all subsequent writeback messages.

Figure 5.7 gives the M-engine rules, in which only the sources of suspended messages are recorded. The transient memory state T[dir,dir_1] means that the memory has sent purge requests to cache sites dir and has received writeback messages from cache sites dir_1. The memory uses FlushAck messages to acknowledge all suspended writeback messages. It is worth pointing out that, if the memory remembers which writeback has been used to update the memory, it can acknowledge that writeback message via a WbAck message to allow the cache to retain a clean copy.

5.4 Soundness Proof of the WP Protocol

In this section, we prove the soundness of WP by showing that CRF can simulate WP. We define a mapping function from WP to CRF, and show that any imperative rule of WP can be simulated in CRF with respect to the mapping function. The soundness of WP follows from the fact that all the WP rules can be derived from the imperative and directive rules of WP.

The soundness proof is given under the assumption that messages can be reordered arbitrarily in incoming and outgoing queues. Obviously, the soundness property cannot be compromised in the presence of specific reordering restrictions such as FIFO message passing. We first present the invariants that will be used throughout the proof; all the invariants are given in the context of the imperative rules of WP.

5.4.1 Some Invariants of WP

Lemma 15 includes two invariants that describe the correspondence between memory states, cache states and messages in transit. Invariant (1) means that the directory shows that an address is cached in a cache site if and only if the address is cached in the Clean or Dirty state in the cache, or a Cache or WbAck message is in transit from the memory to the cache, or a Purge or Wb message is in transit from the cache to the memory. Furthermore, a clean cell always contains the same value as the memory cell. Invariant (2) describes the message in transit when the cache state shows that a writeback operation is being performed. It ensures that a WbAck or FlushAck message can always be processed when it is received.

Lemma 15 Given a WP term s,

$$\begin{aligned}
&(1)\quad \mathrm{Cell}(a,v,\mathrm{T}[id|\text{-},\text{-}]) \in \mathrm{Mem}(s) \quad \Leftrightarrow \\
&\qquad\qquad \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(s) \;\vee\; \mathrm{Cell}(a,\text{-},\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s) \;\vee \\
&\qquad\qquad \mathrm{Msg}(\mathrm{H},id,\mathrm{Cache},a,v) \in \mathrm{MoutCin}_{id}(s) \;\vee\; \mathrm{Msg}(\mathrm{H},id,\mathrm{WbAck},a) \in \mathrm{MoutCin}_{id}(s) \;\vee \\
&\qquad\qquad \mathrm{Msg}(id,\mathrm{H},\mathrm{Purge},a) \in \mathrm{MinCout}_{id}(s) \;\vee\; \mathrm{Msg}(id,\mathrm{H},\mathrm{Wb},a,\text{-}) \in \mathrm{MinCout}_{id}(s) \\
&(2)\quad \mathrm{Cell}(a,v,\mathrm{WbPending}) \in \mathrm{Cache}_{id}(s) \quad \Leftrightarrow \\
&\qquad\qquad \mathrm{Msg}(id,\mathrm{H},\mathrm{Wb},a,v) \in \mathrm{MinCout}_{id}(s) \;\vee\; \mathrm{Cell}(a,\text{-},\mathrm{T}[\text{-},(id,v)|\text{-}]) \in \mathrm{Mem}(s) \;\vee \\
&\qquad\qquad \mathrm{Msg}(\mathrm{H},id,\mathrm{WbAck},a) \in \mathrm{MoutCin}_{id}(s) \;\vee\; \mathrm{Msg}(\mathrm{H},id,\mathrm{FlushAck},a) \in \mathrm{MoutCin}_{id}(s)
\end{aligned}$$

Lemma 16 implies that, if a Cache message is on its way to a cache, then the address is not cached in the cache or a FlushAck message is on its way to the cache. This ensures that a

Cache message can always be processed eventually.

Lemma 16 Given a WP term s,

$$\begin{array}{l} \text{Msg}(\text{H},id,\text{Cache},a,\text{-}) \in \text{MoutCin}_{id}(s) \quad \Rightarrow \\ \qquad a \notin \text{Cache}_{id}(s) \ \vee \ \text{Msg}(\text{H},id,\text{FlushAck},a) \in \text{MoutCin}_{id}(s) \end{array}$$

Lemma 17 means that at any time, there can be at most one outstanding message on each address between the same source and destination, except for Cache and FlushAck messages.

Lemma 17 Given a WP term s,

$$\begin{array}{l} msg_1 \in s \ \wedge \ msg_2 \in s \quad \Rightarrow \\ \qquad \text{Src}(msg_1) \neq \text{Src}(msg_2) \ \vee \ \text{Dest}(msg_1) \neq \text{Dest}(msg_2) \ \vee \ \text{Addr}(msg_1) \neq \text{Addr}(msg_2) \ \vee \\ \qquad (\text{Cmd}(msg_1) = \text{Cache} \ \wedge \ \text{Cmd}(msg_2) = \text{FlushAck}) \ \vee \\ \qquad (\text{Cmd}(msg_1) = \text{FlushAck} \ \wedge \ \text{Cmd}(msg_2) = \text{Cache}) \end{array}$$

Proof (Lemmas 15, 16 and 17) The proof is based on induction on rewriting steps. The invariants hold trivially for the initial term where all caches and queues are empty. It can be shown by checking each rule that, if the invariants hold for a term, then they still hold after the term is rewritten according to that rule. □

5.4.2 Mapping from WP to CRF

We define a mapping function that maps terms of WP to terms of CRF. For WP terms in which all message queues are empty, it is straightforward to find the corresponding CRF terms. There is a one-to-one correspondence between these drained terms of WP and the terms of CRF. For WP terms that contain non-empty message queues, we apply a set of draining rules to extract all the messages from the queues.

We use backward draining for Wb messages and forward draining for all other messages (note that forwarding draining of Wb messages would lead to non-deterministic drained terms when there are multiple writeback messages regarding the same address). Consequently, all the Cache, WbAck, FlushAck and Wb messages will be drained at cache sites, while all the Purge messages will be drained at the memory.

Backward Rules: The *Backward-M-Receive-Wb* rule allows the memory to extract a Wb message from the suspended message buffer and place it back to the incoming queue. The *Backward-Message-Cache-to-Mem-for-Wb* rule moves a Wb message from the memory's incoming queue back to the source cache's outgoing queue. The *Backward-C-Send-Wb* rule allows a cache to reclaim a Wb message from its outgoing queue and recover the cache state.

Backward-M-Receive-Wb Rule

$$\begin{array}{ll} & \text{Msite}(\text{Cell}(a,v_1,\text{T}[dir,sm|(id,v)]) \,|\, mem,\ in,\ out) \\ \to & \text{Msite}(\text{Cell}(a,v_1,\text{T}[id|dir,sm]) \,|\, mem,\ \text{Msg}(id,\text{H},\text{Wb},a,v) \odot in,\ out) \end{array}$$

Backward-Message-Cache-to-Mem-for-Wb Rule

$$\begin{array}{ll}
 & \mathrm{Sys}(\mathrm{Msite}(\mathit{mem},\ \mathit{min}\odot \mathit{msg},\ \mathit{mout}),\ \mathrm{Site}(\mathit{id},\ \mathit{cache},\ \mathit{cin},\ \mathit{cout},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc})\ |\ \mathit{sites}) \\
 & \qquad\qquad \textit{if} \quad \mathrm{Src}(\mathit{msg}) = \mathit{id} \ \wedge\ \mathrm{Cmd}(\mathit{msg}) = \mathrm{Wb} \\
\rightarrow & \mathrm{Sys}(\mathrm{Msite}(\mathit{mem},\ \mathit{min},\ \mathit{mout}),\ \mathrm{Site}(\mathit{id},\ \mathit{cache},\ \mathit{cin},\ \mathit{msg}\otimes \mathit{cout},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc})\ |\ \mathit{sites})
\end{array}$$

Backward-C-Send-Wb Rule

$$\begin{array}{ll}
 & \mathrm{Site}(\mathit{id},\ \mathrm{Cell}(a,v,\mathrm{WbPending})\ |\ \mathit{cache},\ \mathit{in},\ \mathit{out}\otimes \mathrm{Msg}(\mathit{id},\mathrm{H},\mathrm{Wb},a,v),\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc}) \\
\rightarrow & \mathrm{Site}(\mathit{id},\ \mathrm{Cell}(a,v,\mathrm{Dirty})\ |\ \mathit{cache},\ \mathit{in},\ \mathit{out},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc})
\end{array}$$

The backward rules above will be used for backward draining of Wb messages. They are the backward version of the *M-Receive-Wb*, *Message-Cache-to-Mem* (for Wb messages) and *C-Send-Wb* rules, respectively. It is trivial to show that Invariants 15, 16 and 17 still hold in the presence of the backward rules.

In addition to the backward rules, the draining rules also contain some WP rules that are needed to drain Cache, WbAck, FlushAck and Purge messages. Furthermore, we need to tailor the cache-to-memory message passing rule to disallow Wb messages to flow to the memory throughout the draining process. The following *Message-Cache-to-Mem-for-Purge* rule is a restricted version of the *Message-Cache-to-Mem* rule.

Message-Cache-to-Mem-for-Purge Rule

$$\begin{array}{ll}
 & \mathrm{Sys}(\mathrm{Msite}(\mathit{mem},\ \mathit{min},\ \mathit{mout}),\ \mathrm{Site}(\mathit{id},\ \mathit{cache},\ \mathit{cin},\ \mathit{msg}\otimes \mathit{cout},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc})\ |\ \mathit{sites}) \\
 & \qquad\qquad \textit{if} \quad \mathrm{Dest}(\mathit{msg}) = \mathrm{H} \ \wedge\ \mathrm{Cmd}(\mathit{msg}) = \mathrm{Purge} \\
\rightarrow & \mathrm{Sys}(\mathrm{Msite}(\mathit{mem},\ \mathit{min}\odot \mathit{msg},\ \mathit{mout}),\ \mathrm{Site}(\mathit{id},\ \mathit{cache},\ \mathit{cin},\ \mathit{cout},\ \mathit{pmb},\ \mathit{mpb},\ \mathit{proc})\ |\ \mathit{sites})
\end{array}$$

Definition 18 (Draining Rules) Given a WP term s, the drained term $\mathrm{dr}(s)$ is the normal form of s with respect to the following draining rules:

$$\begin{array}{lll}
D & \equiv & \{ \quad \textit{C-Receive-Cache},\ \textit{C-Receive-WbAck}, \\
 & & \quad\ \ \textit{C-Receive-FlushAck},\ \textit{M-Receive-Purge}, \\
 & & \quad\ \ \textit{Backward-M-Receive-Wb},\ \textit{Backward-C-Send-Wb}, \\
 & & \quad\ \ \textit{Message-Mem-to-Cache},\ \textit{Message-Cache-to-Mem-for-Purge}, \\
 & & \quad\ \ \textit{Backward-Message-Cache-to-Mem-for-Wb} \quad \}
\end{array}$$

Lemma 19 D is strongly terminating and confluent, that is, rewriting a WP term with respect to the draining rules always terminates and reaches the same normal form, regardless of the order in which the rules are applied.

Proof The termination is obvious because according to the draining rules, Cache, WbAck, FlushAck and Wb messages can only flow from memory to caches, and Purge messages can only flow from caches to memory. The confluence follows from the fact that the draining rules do not interfere with each other. □

Lemma 20 ensures that the processor-to-memory buffers, the memory-to-processor buffers and the processors all remain unchanged after the draining rules are applied. Lemma 21 ensures that the message queues all become empty in a drained term. This can be proved according to Lemma 15, which guarantees that a Cache, WbAck, FlushAck or Wb message can be consumed at the cache, and a Purge message can be consumed at the memory.

Lemma 20 Given a WP term s,

(1) $\mathrm{Pmb}_{id}(s) \;=\; \mathrm{Pmb}_{id}(\mathrm{dr}(s))$
(2) $\mathrm{Mpb}_{id}(s) \;=\; \mathrm{Mpb}_{id}(\mathrm{dr}(s))$
(3) $\mathrm{Proc}_{id}(s) \;=\; \mathrm{Proc}_{id}(\mathrm{dr}(s))$

Lemma 21 Given a WP term s,

(1) $\mathrm{Min}(\mathrm{dr}(s)) \;=\; \epsilon$
(2) $\mathrm{Mout}(\mathrm{dr}(s)) \;=\; \epsilon$
(3) $\mathrm{Cin}_{id}(\mathrm{dr}(s)) \;=\; \epsilon$
(4) $\mathrm{Cout}_{id}(\mathrm{dr}(s)) \;=\; \epsilon$

The draining rules have no impact on the value of a memory cell. A cache cell in a stable state remains unaffected. An uncached address remains uncached provided that no Cache message is drained. Lemma 22 captures these properties.

Lemma 22 Given a WP term s,

(1) $\mathrm{Cell}(a,v,\mathrm{T}[\text{-},\text{-}]) \in \mathrm{Mem}(s) \;\Rightarrow\; \mathrm{Cell}(a,v,\mathrm{T}[\text{-},\epsilon]) \in \mathrm{Mem}(\mathrm{dr}(s))$
(2) $\mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(s) \;\Rightarrow\; \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s))$
(3) $\mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s) \;\Rightarrow\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s))$
(4) $a \notin \mathrm{Cache}_{id}(s) \;\wedge\; \mathrm{Msg}(\mathrm{H},id,\mathrm{Cache},a,\text{-}) \notin \mathrm{MoutCin}_{id}(s_1) \;\Rightarrow\; a \notin \mathrm{Cache}_{id}(\mathrm{dr}(s))$

Lemma 23 ensures that in a drained term, an address will be cached in the Clean state if a Cache or WbAck message is drained; an address will be cached in the Dirty state if a Wb message is drained; an address will be uncached if a Purge or FlushAck message is drained. The proof follows from Lemma 15 and the draining rules (note that messages can be drained in any order because of the confluence of the draining rules).

Lemma 23 Given a WP term s,

(1) $\mathrm{Msg}(\mathrm{H},id,\mathrm{Cache},a,v) \in \mathrm{MoutCin}_{id}(s) \;\Rightarrow\; \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s))$
(2) $\mathrm{Msg}(id,\mathrm{H},\mathrm{Purge},a) \in \mathrm{MinCout}_{id}(s) \;\Rightarrow\; a \notin \mathrm{Cache}_{id}(\mathrm{dr}(s))$
(3) $\mathrm{Msg}(id,\mathrm{H},\mathrm{Wb},a,v) \in \mathrm{MinCout}_{id}(s) \;\Rightarrow\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s))$
(4) $\mathrm{Cell}(a,v,\mathrm{T}[\text{-},(id,v)|\text{-}]) \in \mathrm{Mem}(s) \;\Rightarrow\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s))$
(5) $\mathrm{Msg}(\mathrm{H},id,\mathrm{WbAck},a) \in \mathrm{MoutCin}_{id}(s) \;\Rightarrow\; \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s))$
(6) $\mathrm{Msg}(\mathrm{H},id,\mathrm{FlushAck},a) \in \mathrm{MoutCin}_{id}(s) \;\wedge\; \mathrm{Msg}(\mathrm{H},id,\mathrm{Cache},a,\text{-}) \notin \mathrm{MoutCin}_{id}(s)$
$\Rightarrow\; a \notin \mathrm{Cache}_{id}(\mathrm{dr}(s))$

Definition 24 (Mapping from WP to CRF) Given a WP term s, the corresponding CRF term $\mathrm{f}(s)$ is the drained term $\mathrm{dr}(s)$ with all message queues and cache identifiers removed.

It is obvious that the mapping function maps the initial WP term (with all caches and network queues empty) to the initial CRF term (with all semantic caches empty). For any WP term, the mapping function guarantees that it is mapped to a legal CRF term (this follows trivially from the simulation theorem given below).

5.4.3 Simulation of WP in CRF

Theorem 25 (CRF Simulates WP) Given WP terms s_1 and s_2,

$$s_1 \rightarrow s_2 \quad \text{in WP} \qquad \Rightarrow \qquad \mathrm{f}(s_1) \twoheadrightarrow \mathrm{f}(s_2) \quad \text{in CRF}$$

Proof The proof is based on a case analysis on the imperative rule used in the rewriting of '$s_1 \rightarrow s_2$' in WP. Let a be the address and id the cache identifier.

Imperative Processor Rules

- If Rule IP1 (*Loadl-on-Clean*) applies, then

$$\begin{array}{ll} & \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(s_1) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(s_2) \\ \Rightarrow & \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 22}) \\ \Rightarrow & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Loadl}) \end{array}$$

- If Rule IP2 (*Loadl-on-Dirty*) applies, then

$$\begin{array}{ll} & \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_1) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_2) \\ \Rightarrow & \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 22}) \\ \Rightarrow & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Loadl}) \end{array}$$

- If Rule IP3 (*Storel-on-Clean*) applies, then

$$\begin{array}{ll} & \mathrm{Cell}(a,\text{-},\mathrm{Clean}) \in \mathrm{Cache}_{id}(s_1) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_2) \\ \Rightarrow & \mathrm{Cell}(a,\text{-},\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 22}) \\ \Rightarrow & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Storel}) \end{array}$$

- If Rule IP4 (*Storel-on-Dirty*) applies, then

$$\begin{array}{ll} & \mathrm{Cell}(a,\text{-},\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_1) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_2) \\ \Rightarrow & \mathrm{Cell}(a,\text{-},\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 22}) \\ \Rightarrow & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Storel}) \end{array}$$

- If Rule IP5 (*Commit-on-Clean*) applies, then

$$\begin{array}{ll} & \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(s_1) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(s_2) \\ \Rightarrow & \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 22}) \\ \Rightarrow & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Commit}) \end{array}$$

- If Rule IP6 (*Commit-on-Invalid*) applies, and if $\mathrm{Msg}(\mathrm{H},id,\mathrm{Cache},a,\text{-}) \notin \mathrm{MoutCin}_{id}(s_1)$, then

$$\begin{array}{ll} & a \notin \mathrm{Cache}_{id}(s_1) \;\wedge\; a \notin \mathrm{Cache}_{id}(s_2) \\ \Rightarrow & a \notin \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; a \notin \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 22}) \\ \Rightarrow & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Commit}) \end{array}$$

- If Rule IP6 (*Commit-on-Invalid*) applies, and if $\mathrm{Msg}(\mathrm{H},id,\mathrm{Cache},a,v) \in \mathrm{MoutCin}_{id}(s_1)$, then

$$\begin{array}{ll} & a \notin \mathrm{Cache}_{id}(s_1) \;\wedge\; a \notin \mathrm{Cache}_{id}(s_2) \\ \Rightarrow & \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 23}) \\ \Rightarrow & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Commit}) \end{array}$$

- If Rule IP7 (*Reconcile-on-Dirty*) applies, then

$$\begin{array}{ll} & \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_1) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s_2) \\ \Rightarrow & \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_1)) \;\wedge\; \mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(\mathrm{dr}(s_2)) \quad (\textit{Lemma 22}) \\ \Rightarrow & \mathrm{f}(s_1) \rightarrow \mathrm{f}(s_2) \qquad (\textit{CRF-Reconcile}) \end{array}$$

- If Rule IP8 (*Reconcile-on-Clean*) applies, then

$$\begin{array}{lll}
 & \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(s_1) \ \wedge\ \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(s_2) & \\
\Rightarrow & \text{Cell}(a,v,\text{T}[\text{-},\text{-}]) \in \text{Mem}(s_1) \ \wedge\ \text{Cell}(a,v,\text{T}[\text{-},\text{-}]) \in \text{Mem}(s_2) & (\textit{Lemma 15}) \\
\Rightarrow & \text{Cell}(a,v,\text{T}[\text{-},\epsilon]) \in \text{Mem}(\text{dr}(s_1)) \ \wedge\ \text{Cell}(a,v,\text{T}[\text{-},\epsilon]) \in \text{Mem}(\text{dr}(s_2)) \ \wedge & (\textit{Lemma 22}) \\
 & \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s_1)) \ \wedge\ \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s_2)) & (\textit{Lemma 22}) \\
\Rightarrow & \text{f}(s_1) \twoheadrightarrow \text{f}(s_2) \qquad (\textit{CRF-Purge, CRF-Reconcile \& CRF-Cache}) &
\end{array}$$

- If Rule IP9 (*Reconcile-on-Invalid*) applies, and if $\text{Msg}(\text{H},id,\text{Cache},a,\text{-}) \notin \text{MoutCin}_{id}(s_1)$, then

$$\begin{array}{lll}
 & a \notin \text{Cache}_{id}(s_1) \ \wedge\ a \notin \text{Cache}_{id}(s_2) & \\
\Rightarrow & a \notin \text{Cache}_{id}(\text{dr}(s_1)) \ \wedge\ a \notin \text{Cache}_{id}(\text{dr}(s_2)) & (\textit{Lemma 22}) \\
\Rightarrow & \text{f}(s_1) \rightarrow \text{f}(s_2) \qquad (\textit{CRF-Reconcile}) &
\end{array}$$

- If Rule IP9 (*Reconcile-on-Invalid*) applies, and if $\text{Msg}(\text{H},id,\text{Cache},a,v) \in \text{MoutCin}_{id}(s_1)$, then

$$\begin{array}{lll}
 & a \notin \text{Cache}_{id}(s_1) \ \wedge\ a \notin \text{Cache}_{id}(s_2) & \\
\Rightarrow & \text{Cell}(a,v,\text{T}[\text{-},\text{-}]) \in \text{Mem}(s_1) \ \wedge\ \text{Cell}(a,v,\text{T}[\text{-},\text{-}]) \in \text{Mem}(s_2) & (\textit{Lemma 15}) \\
\Rightarrow & \text{Cell}(a,v,\text{T}[\text{-},\epsilon]) \in \text{Mem}(\text{dr}(s_1)) \ \wedge\ \text{Cell}(a,v,\text{T}[\text{-},\epsilon]) \in \text{Mem}(\text{dr}(s_2)) \ \wedge & (\textit{Lemma 22}) \\
 & \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s_1)) \ \wedge\ \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s_2)) & (\textit{Lemma 23}) \\
\Rightarrow & \text{f}(s_1) \twoheadrightarrow \text{f}(s_2) \qquad (\textit{CRF-Purge, CRF-Reconcile \& CRF-Cache}) &
\end{array}$$

Imperative C-engine Rules

- If Rule IC1 (*C-Send-Purge*) applies, then

$$\begin{array}{lll}
 & \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(s_1) \ \wedge & \\
 & a \notin \text{Cache}_{id}(s_2) \ \wedge\ \text{Msg}(id,\text{H},\text{Purge},a) \in \text{Cout}_{id}(s_2) & \\
\Rightarrow & \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s_1)) \ \wedge & (\textit{Lemma 22}) \\
 & a \notin \text{Cache}_{id}(\text{dr}(s_2)) & (\textit{Lemma 23}) \\
\Rightarrow & \text{f}(s_1) \rightarrow \text{f}(s_2) \qquad (\textit{CRF-Purge}) &
\end{array}$$

Imperative M-engine Rules

- If Rule IM1 (*M-Send-Cache*) applies, and if $\text{Msg}(\text{H},id,\text{FlushAck},a) \notin \text{MoutCin}_{id}(s_1)$, then

$$\begin{array}{lll}
 & \text{Cell}(a,v,\text{T}[dir,\epsilon]) \in \text{Mem}(s_1),\ id \notin dir & \\
 & \text{Cell}(a,v,\text{T}[id|dir,\epsilon]) \in \text{Mem}(s_2),\ \text{Msg}(\text{H},id,\text{Cache},a,v) \in \text{Mout}(s_2) & \\
\Rightarrow & a \notin s_1,\ \text{Msg}(\text{H},id,\text{Cache},a,\text{-}) \notin \text{Mout}(s_1) + \text{Cin}_{id}(s_1) & (\textit{Lemma 15}) \\
\Rightarrow & a \notin \text{dr}(s_1) & (\textit{Lemma 22}) \\
\Rightarrow & \text{Cell}(a,v,\text{T}[\text{-},\epsilon]) \in \text{Mem}(\text{dr}(s_1)),\ \text{Mem}(\text{dr}(s_2)) & (\textit{Lemma 22}) \\
 & \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s_2)) & (\textit{Lemma 23}) \\
\Rightarrow & \text{f}(s_1) \rightarrow \text{f}(s_2) \qquad (\textit{CRF-Cache}) &
\end{array}$$

- If Rule IM1 (*M-Send-Cache*) applies, and if $\text{Msg}(\text{H},id,\text{FlushAck},a) \in \text{MoutCin}_{id}(s_1)$, then

$$\begin{array}{lll}
 & \text{Cell}(a,v,\text{T}[dir,\epsilon]) \in \text{Mem}(s_1),\ id \notin dir & \\
 & \text{Cell}(a,v,\text{T}[id|dir,\epsilon]) \in \text{Mem}(s_2),\ \text{Msg}(\text{H},id,\text{Cache},a,v) \in \text{Mout}(s_2) & \\
\Rightarrow & \text{Msg}(\text{H},id,\text{Cache},a,\text{-}) \notin \text{Mout}(s_1) + \text{Cin}_{id}(s_1) & (\textit{Lemma 15}) \\
\Rightarrow & a \notin \text{dr}(s_1) & (\textit{Lemma 22}) \\
\Rightarrow & \text{Cell}(a,v,\text{T}[\text{-},\epsilon]) \in \text{Mem}(\text{dr}(s_1)),\ \text{Mem}(\text{dr}(s_2)) & (\textit{Lemma 22}) \\
 & \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s_2)) & (\textit{Lemma 23}) \\
\Rightarrow & \text{f}(s_1) \rightarrow \text{f}(s_2) \qquad (\textit{CRF-Cache}) &
\end{array}$$

WP Imperative Rule	**CRF Rules**
IP1 (*Loadl-on-Clean*)	*CRF-Loadl*
IP2 (*Loadl-on-Dirty*)	*CRF-Loadl*
IP3 (*Storel-on-Clean*)	*CRF-Storel*
IP4 (*Storel-on-Dirty*)	*CRF-Storel*
IP5 (*Commit-on-Clean*)	*CRF-Commit*
IP6 (*Commit-on-Invalid*)	*CRF-Commit*
IP7 (*Reconcile-on-Dirty*)	*CRF-Reconcile*
IP8 (*Reconcile-on-Clean*)	*CRF-Purge* + *CRF-Reconcile* + *CRF-Cache*
IP9 (*Reconcile-on-Invalid*)	*CRF-Reconcile*, or *CRF-Purge* + *CRF-Reconcile* + *CRF-Cache*
IC1 (*C-Send-Purge*)	*CRF-Purge*
IC2 (*C-Send-Wb*)	ϵ
IC3 (*C-Receive-WbAck*)	ϵ
IC4 (*C-Receive-FlushAck*)	ϵ
IC5 (*C-Receive-Cache*)	ϵ
IM1 (*M-Send-Cache*)	*CRF-Cache*
IM2 (*M-Receive-Purge*)	ϵ
IM3 (*M-Receive-Wb*)	ϵ
IM4 (*M-Send-FlushAck*)	*CRF-Writeback* + *CRF-Purge*
IM5 (*M-Send-WbAck*)	*CRF-Writeback*
Message-Cache-to-Mem	ϵ
Message-Mem-to-Cache	ϵ

Figure 5.8: Simulation of WP in CRF

- If Rule IM4 (*M-Send-FlushAck*) applies, then

$$
\begin{array}{ll}
 & \text{Cell}(a,\text{-},\text{T}[\epsilon,(id,v)|sm]) \in \text{Mem}(s_1) \;\wedge \\
 & \text{Cell}(a,v,\text{T}[\epsilon,sm]) \in \text{Mem}(s_2) \;\wedge\; \text{Msg}(\text{H},id,\text{FlushAck},a) \in \text{Mout}(s_2) \\
\Rightarrow & \text{Msg}(\text{H},id,\text{Cache},a,\text{-}) \notin \text{MoutCin}_{id}(s_2) \qquad (\textit{Lemma 15}) \\
\Rightarrow & \text{Cell}(a,v,\text{Dirty}) \in \text{Cache}_{id}(\text{dr}(s_1)) \;\wedge \qquad (\textit{Lemma 23}) \\
 & a \notin \text{dr}(s_2) \;\wedge \qquad (\textit{Lemma 23}) \\
 & \text{Cell}(a,v,\text{T}[\text{-},\epsilon]) \in \text{Mem}(\text{dr}(s_2)) \qquad (\textit{Lemma 22}) \\
\Rightarrow & \text{f}(s_1) \twoheadrightarrow \text{f}(s_2) \qquad (\textit{CRF-Writeback \& CRF-Purge})
\end{array}
$$

- If Rule IM5 (*M-Send-WbAck*) applies, then

$$
\begin{array}{ll}
 & \text{Cell}(a,\text{-},\text{T}[\epsilon,(id,v)]) \in \text{Mem}(s_1) \;\wedge \\
 & \text{Cell}(a,v,\text{T}[id,\epsilon]) \in \text{Mem}(s_2) \;\wedge\; \text{Msg}(\text{H},id,\text{WbAck},a) \in \text{Mout}(s_2) \\
\Rightarrow & \text{Cell}(a,v,\text{Dirty}) \in \text{Cache}_{id}(\text{dr}(s_1)) \;\wedge \qquad (\textit{Lemma 23}) \\
 & \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(\text{dr}(s_2)) \;\wedge \qquad (\textit{Lemma 23}) \\
 & \text{Cell}(a,v,\text{T}[\text{-},\epsilon]) \in \text{Mem}(\text{dr}(s_2)) \qquad (\textit{Lemma 22}) \\
\Rightarrow & \text{f}(s_1) \rightarrow \text{f}(s_2) \qquad (\textit{CRF-Writeback})
\end{array}
$$

Draining Rules

- If Rule IC2 (*C-Send-Wb*), IC3 (*C-Receive-WbAck*), IC4 (*C-Receive-FlushAck*), IC5 (*C-Receive-Cache*), IM2 (*M-Receive-Purge*), IM3 (*M-Receive-Wb*), *Message-Cache-to-Mem* or *Message-Mem-to-Cache* applies, then

$$\text{f}(s_1) = \text{f}(s_2) \qquad (\textit{Since the rule or its backward version is a draining rule})$$

Figure 5.8 summarizes the simulation proof. □

5.4.4 Soundness of WP

The WP protocol defined in Figure 5.5 consists of integrated rules that can be derived from the imperative and directive rules of WP. The imperative rules are given in Section 5.2. In the remainder of this section, we give the directive rules and show the derivation of WP rules. A directive rule involves generating or discarding a directive message, which can be used to specify conditions under which an imperative action should be invoked.

C-Send-CacheReq Rule

$\quad$ Site(id, $cache$, in, out, pmb, mpb, $proc$)

$\rightarrow$ Site(id, $cache$, in, $out \otimes$ Msg(id,H,CacheReq,a), pmb, mpb, $proc$)

C-Receive-PurgeReq Rule

$\quad$ Site(id, $cache$, Msg(H,id,PurgeReq,a) $\odot$ in, out, pmb, mpb, $proc$)

$\rightarrow$ Site(id, $cache$, in, out, pmb, mpb, $proc$)

M-Send-PurgeReq Rule

$\quad$ Msite(mem, in, out)

$\rightarrow$ Msite(mem, in, $out \otimes$ Msg(H,id,PurgeReq,a))

M-Receive-CacheReq Rule

$\quad$ Msite(mem, Msg(id,H,CacheReq,a) $\odot$ in, out)

$\rightarrow$ Msite(mem, in, out)

Figure 5.9 gives the imperative and directive rules used in the derivation for each WP rule (a rule marked with '$*$' may be applied zero or many times). In the derivation, the CachePending state used in the integrated rules is mapped to the Invalid state of the imperative rules, and the C[dir] state used in the integrated rules is mapped to the T[dir,ϵ] state of the imperative rules. For example, consider Rule MM5 that deals with an incoming writeback message. It involves applying the imperative *M-Receive-Wb* rule to suspend the writeback message, and the directive *M-Send-PurgeReq* rule to generate purge requests.

A directive rule by itself cannot modify any system state that may affect soundness. Therefore, it suffices to verify the soundness of the protocol with respect to the imperative rules, rather than the integrated rules. This can dramatically simplify the verification since the number of imperative rules is much smaller than the number of integrated rules.

5.5 Liveness Proof of the WP Protocol

In this section, we prove the liveness of WP by showing that an instruction can always be completed so that each processor can make progress. That is, whenever a processor intends to execute a memory instruction, the cache cell will be brought to an appropriate state so that the instruction can be retired. The lemmas and theorems in this section are given in the context of the integrated rules of WP, which involve both imperative and directive messages. We assume FIFO message passing and proper buffer management as described in Section 5.3.

WP Rule	**WP Imperative & Directive Rules**
P1	*Loadl-on-Clean*
P2	*Loadl-on-Dirty*
P3	ϵ
P4	ϵ
P5	*C-Send-CacheReq*
P6	*Storel-on-Clean*
P7	*Storel-on-Dirty*
P8	ϵ
P9	ϵ
P10	*C-Send-CacheReq*
P11	*Commit-on-Clean*
P12	*C-Send-Wb*
P13	ϵ
P14	ϵ
P15	*Commit-on-Invalid*
P16	*Reconcile-on-Clean*
P17	*Reconcile-on-Dirty*
P18	ϵ
P19	ϵ
P20	*Reconcile-on-Invalid*
VC1	*C-Send-Purge*
VC2	*C-Send-Wb*
VC3	*C-Send-CacheReq*
MC1	*C-Receive-Cache*
MC2	*C-Receive-Cache*
MC3	*C-Receive-WbAck*
MC4	*C-Receive-FlushAck*
MC5	*C-Receive-PurgeReq* + *C-Send-Purge*
MC6	*C-Receive-PurgeReq* + *C-Send-Wb*
MC7	*C-Receive-PurgeReq*
MC8	*C-Receive-PurgeReq*
MC9	*C-Receive-PurgeReq*
VM1	*M-Send-Cache*
VM2	*M-Send-PurgeReq**
MM1	*M-Receive-CacheReq* + *M-Send-Cache*
MM2	*M-Receive-CacheReq*
MM3	ϵ
MM4	*M-Receive-CacheReq*
MM5	*M-Receive-Wb* + *M-Send-PurgeReq**
MM6	*M-Receive-Wb*
MM7	*M-Receive-Purge*
MM8	*M-Receive-Purge*
MM9	*M-Send-FlushAck*
MM10	*M-Send-WbAck*
MM11	ϵ

Figure 5.9: Derivation of WP from Imperative & Directive Rules

5.5.1 Some Invariants of WP

Lemma 26 is identical to Lemma 15, except that the T[dir,ϵ] state used in the imperative rules is replaced by the C[dir] and T[dir,ϵ] states used in the integrated rules.

Lemma 26 Given a WP term s,

(1) $$\begin{aligned}&\text{Cell}(a,v,\text{C}[id|\text{-}]) \in \text{Mem}(s) \;\vee\; \text{Cell}(a,v,\text{T}[id|\text{-},\text{-}]) \in \text{Mem}(s) \quad \Leftrightarrow\\ &\quad \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(s) \;\vee\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(s) \;\vee\\ &\quad \text{Msg}(\text{H},id,\text{Cache},a,v) \in \text{MoutCin}_{id}(s) \;\vee\; \text{Msg}(\text{H},id,\text{WbAck},a) \in \text{MoutCin}_{id}(s) \;\vee\\ &\quad \text{Msg}(id,\text{H},\text{Purge},a) \in \text{MinCout}_{id}(s) \;\vee\; \text{Msg}(id,\text{H},\text{Wb},a,\text{-}) \in \text{MinCout}_{id}(s)\end{aligned}$$

(2) $$\begin{aligned}&\text{Cell}(a,v,\text{WbPending}) \in \text{Cache}_{id}(s) \quad \Leftrightarrow\\ &\quad \text{Msg}(id,\text{H},\text{Wb},a,v) \in \text{MinCout}_{id}(s) \;\vee\; \text{Cell}(a,\text{-},\text{T}[\text{-},(id,v)|\text{-}]) \in \text{Mem}(s) \;\vee\\ &\quad \text{Msg}(\text{H},id,\text{WbAck},a) \in \text{MoutCin}_{id}(s) \;\vee\; \text{Msg}(\text{H},id,\text{FlushAck},a) \in \text{MoutCin}_{id}(s)\end{aligned}$$

Lemma 27 includes invariants regarding message generation and processing. Invariants (1)-(2) ensure that when a Loadl or Storel instruction is performed on an uncached address, the cache will contain a clean cell for the address, or issue a CacheReq message and set the cache state to CachePending. Invariant (3) ensures that when a Commit instruction is performed on a dirty cell, the cache will issue a Wb message and set the cache state to WbPending. The proof follows from the weak fairness of Rules P5, P10 and P12.

Invariants (4)-(8) ensure that when a cache receives a Cache or WbAck message, it will set the cache state to Clean; when a cache receives a FlushAck message, it will purge the address; when a cache receives a PurgeReq message, it will send a Purge or Wb message depending on whether a clean or dirty copy is cached for the address. The proof follows from the weak fairness of Rules MC1, MC2, MC3, MC4, MC5 and MC6.

Invariants (9)-(11) ensure that when the memory receives a Purge message, it will remove the cache identifier from the directory; when the memory receives a Wb message, it will suspend the message; when the directory becomes empty, the memory will resume a suspended message. The proof follows from the weak fairness of Rules MM5, MM6, MM7, MM8 and MM9. Notation dir_{-id} represents a directory that does not contains identifier id.

Invariants (12)-(13) ensure that each outgoing message will be delivered to its destination's incoming queue. This can be proved by simple induction on the number of preceding messages in the outgoing queue (note that the message passing rules are weakly fair).

Lemma 27 Given a WP sequence σ,

(1) $$\begin{aligned}&\langle t,\text{Loadl}(a)\rangle \in \text{Pmb}_{id}(\sigma) \;\wedge\; a \notin \text{Cache}_{id}(\sigma) \quad \rightsquigarrow\\ &\quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee\\ &\quad (\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \;\wedge\; \text{Msg}(id,\text{H},\text{CacheReq},a) \in \text{Cout}_{id}(\sigma))\end{aligned}$$

(2) $$\begin{aligned}&\langle t,\text{Storel}(a,\text{-})\rangle \in \text{Pmb}_{id}(\sigma) \;\wedge\; a \notin \text{Cache}_{id}(\sigma) \quad \rightsquigarrow\\ &\quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee\\ &\quad (\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \;\wedge\; \text{Msg}(id,\text{H},\text{CacheReq},a) \in \text{Cout}_{id}(\sigma))\end{aligned}$$

(3) $$\begin{aligned}&\langle t,\text{Commit}(a)\rangle \in \text{Pmb}_{id}(\sigma) \;\wedge\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \quad \rightsquigarrow\\ &\quad \text{Cell}(a,\text{-},\text{WbPending}) \in \text{Cache}_{id}(\sigma) \;\wedge\; \text{Msg}(id,\text{H},\text{Wb},a,\text{-}) \in \text{Cout}_{id}(\sigma)\end{aligned}$$

(4) $\text{Msg}(\text{H},id,\text{Cache},a,\text{-})_{\uparrow} \in \text{Cin}_{id}(\sigma) \quad\leadsto\quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)$

(5) $\text{Msg}(\text{H},id,\text{WbAck},a)_{\uparrow} \in \text{Cin}_{id}(\sigma) \quad\leadsto\quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)$

(6) $\text{Msg}(\text{H},id,\text{FlushAck},a)_{\uparrow} \in \text{Cin}_{id}(\sigma) \quad\leadsto\quad a \notin \text{Cache}_{id}(\sigma)$

(7) $\text{Msg}(\text{H},id,\text{PurgeReq},a)_{\uparrow} \in \text{Cin}_{id}(\sigma) \;\wedge\; \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \quad\leadsto$
$\text{Msg}(id,\text{H},\text{Purge},a) \in \text{Cout}_{id}(\sigma)$

(8) $\text{Msg}(\text{H},id,\text{PurgeReq},a)_{\uparrow} \in \text{Cin}_{id}(\sigma) \;\wedge\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \quad\leadsto$
$\text{Msg}(id,\text{H},\text{Wb},a,\text{-}) \in \text{Cout}_{id}(\sigma)$

(9) $\text{Msg}(id,\text{H},\text{Purge},a)_{\uparrow} \in \text{Min}(\sigma) \quad\leadsto$
$\text{Cell}(a,\text{-},\text{C}[dir_{-id}]) \in \text{Mem}(\sigma) \;\vee\; \text{Cell}(a,\text{-},\text{T}[dir_{-id},\text{-}]) \in \text{Mem}(\sigma)$

(10) $\text{Msg}(id,\text{H},\text{Wb},a,\text{-})_{\uparrow} \in \text{Min}(\sigma) \quad\leadsto\quad \text{Cell}(a,\text{-},\text{T}[dir_{-id},(id,\text{-})|\text{-}]) \in \text{Mem}(\sigma)$

(11) $\text{Cell}(a,\text{-},\text{T}[\epsilon,(id,\text{-})|\text{-}]) \in \text{Mem}(\sigma) \quad\leadsto$
$\text{Msg}(\text{H},id,\text{WbAck},a) \in \text{Mout}(\sigma) \;\vee\; \text{Msg}(\text{H},id,\text{FlushAck},a) \in \text{Mout}(\sigma)$

(12) $msg \in \text{Cout}_{id}(\sigma) \;\wedge\; \text{Dest}(msg) = \text{H} \quad\leadsto\quad msg \in \text{Min}(\sigma)$

(13) $msg \in \text{Mout}(\sigma) \;\wedge\; \text{Dest}(msg) = id \quad\leadsto\quad msg \in \text{Cin}_{id}(\sigma)$

Lemma 28 ensures that an incoming Cache, WbAck, FlushAck, PurgeReq, Purge or Wb message will eventually become the first message regarding the address in the incoming queue. This implies that the message will be brought to the front end of the incoming queue so that it has an opportunity to be processed. A more general proposition will be presented in Lemma 31, which guarantees that any incoming message will eventually become the first message in the incoming queue.

Lemma 28 Given a WP sequence σ,

(1) $msg \in \text{Cin}_{id}(\sigma) \quad\leadsto\quad msg_{\uparrow} \in \text{Cin}_{id}(\sigma)$

(2) $msg \in \text{Min}(\sigma) \;\wedge\; (\text{Cmd}(msg) = \text{Purge} \;\vee\; \text{Cmd}(msg) = \text{Wb}) \quad\leadsto\quad msg_{\uparrow} \in \text{Min}(\sigma)$

Proof The proof is based on induction on the number of messages that are in front the message in the incoming queue. It is obvious that the first message in a cache's incoming queue can always be processed because of the weak fairness of the mandatory cache engine rules. At the memory side, the first incoming message can always be processed except that a CacheReq message may need to be stalled. The key observation here is that a CacheReq message followed by a Purge or Wb message cannot be stalled. This is because according to Lemma 26, the directory of the memory state contains the cache identifier, which implies that the CacheReq message can be processed according to Rule MM2 or MM4. □

Lemma 29 ensures that if a memory cell is in a transient state and the directory shows that the address is cached in a cache, then the cache's identifier will be removed from the directory eventually. This guarantees that suspended writeback messages will be resumed.

Lemma 29 Given a WP sequence σ,

$\text{Cell}(a,\text{-},\text{T}[id|\text{-},\text{-}]) \in \text{Mem}(\sigma) \quad\leadsto\quad \text{Cell}(a,\text{-},\text{T}[dir_{-id},\text{-}]) \in \text{Mem}(\sigma)$

Proof We first show some properties that are needed for the proof; all the properties can be verified by simply checking the WP rules.

- The memory sends a PurgeReq message to cache id whenever it changes the state of a memory cell to $\mathrm{T}[id|\text{-},\text{-}]$. Note that the PurgeReq message has no following message when it is issued.
$$\begin{aligned}&\mathrm{Cell}(a,\text{-},\mathrm{T}[id|\text{-},\text{-}]) \notin \mathrm{Mem}(\sigma) \ \wedge\ \bigcirc\, \mathrm{Cell}(a,\text{-},\mathrm{T}[id|\text{-},\text{-}]) \in \mathrm{Mem}(\sigma)\\ &\qquad \Rightarrow \quad \bigcirc\, \mathrm{Msg}(\mathrm{H},id,\mathrm{PurgeReq},a)_{\downarrow} \in \mathrm{Mout}(\sigma)\end{aligned}$$

- The memory cannot remove a cache identifier from a directory unless it receives a Purge or Wb message from the cache site.
$$\begin{aligned}&\mathrm{Cell}(a,\text{-},\mathrm{T}[id|\text{-},\text{-}]) \in \mathrm{Mem}(\sigma) \ \wedge\ \bigcirc\, \mathrm{Cell}(a,\text{-},\mathrm{T}[id|\text{-},\text{-}]) \notin \mathrm{Mem}(\sigma)\\ &\qquad \Rightarrow \quad \mathrm{Msg}(id,\mathrm{H},\mathrm{Purge},a) \in \mathrm{Min}(\sigma) \ \vee\ \mathrm{Msg}(id,\mathrm{H},\mathrm{Wb},a,\text{-}) \in \mathrm{Min}(\sigma)\end{aligned}$$

- When the memory removes a cache identifier from the directory of a memory cell in a transient state, the memory state becomes a transient state in which the directory does not contains the cache identifier.
$$\begin{aligned}&\mathrm{Cell}(a,\text{-},\mathrm{T}[id|\text{-},\text{-}]) \in \mathrm{Mem}(\sigma) \ \wedge\ \bigcirc\, \mathrm{Cell}(a,\text{-},\mathrm{T}[id|\text{-},\text{-}]) \notin \mathrm{Mem}(\sigma)\\ &\qquad \Rightarrow \quad \bigcirc\, \mathrm{Cell}(a,\text{-},\mathrm{T}[dir_{-id},\text{-}]) \in \mathrm{Mem}(\sigma)\end{aligned}$$

- The memory cannot send any message to cache id while the memory state is $\mathrm{T}[id|\text{-},\text{-}]$. Thus, a message that has no following message will remain as the last message.
$$\begin{aligned}&\mathrm{Cell}(a,\text{-},\mathrm{T}[id|\text{-},\text{-}]) \in \mathrm{Mem}(\sigma) \ \wedge\ msg_{\downarrow} \in \mathrm{MoutCin}_{id}(\sigma)\\ &\qquad \Rightarrow \quad \bigcirc\, msg_{\downarrow} \in \mathrm{MoutCin}_{id}(\sigma)\end{aligned}$$

We then prove the lemma under the assumption that the memory's outgoing queue contains a PurgeReq message and the PurgeReq message has no following message.

- According to Theorem-C and Lemma 27,
$$\begin{aligned}&\mathrm{Cell}(a,\text{-},\mathrm{T}[id|\text{-},\text{-}]) \in \mathrm{Mem}(\sigma) \ \wedge\ \mathrm{Msg}(\mathrm{H},id,\mathrm{PurgeReq},a)_{\downarrow} \in \mathrm{Mout}(\sigma)\\ \leadsto\ \ &(\mathrm{Cell}(a,\text{-},\mathrm{T}[id|\text{-},\text{-}]) \in \mathrm{Mem}(\sigma) \ \wedge\ \mathrm{Msg}(\mathrm{H},id,\mathrm{PurgeReq},a)_{\downarrow} \in \mathrm{Cin}_{id}(\sigma)) \ \vee\\ &\mathrm{Cell}(a,\text{-},\mathrm{T}[dir_{-id},\text{-}]) \in \mathrm{Mem}(\sigma)\end{aligned}$$

- According to Theorem-C and Lemma 28,
$$\begin{aligned}&\mathrm{Cell}(a,\text{-},\mathrm{T}[id|\text{-},\text{-}]) \in \mathrm{Mem}(\sigma) \ \wedge\ \mathrm{Msg}(\mathrm{H},id,\mathrm{PurgeReq},a)_{\downarrow} \in \mathrm{Cin}_{id}(\sigma)\\ \leadsto\ \ &(\mathrm{Cell}(a,\text{-},\mathrm{T}[id|\text{-},\text{-}]) \in \mathrm{Mem}(\sigma) \ \wedge\ \mathrm{Msg}(\mathrm{H},id,\mathrm{PurgeReq},a)_{\updownarrow} \in \mathrm{Cin}_{id}(\sigma)) \ \vee\\ &\mathrm{Cell}(a,\text{-},\mathrm{T}[dir_{-id},\text{-}]) \in \mathrm{Mem}(\sigma)\end{aligned}$$

- According to Lemmas 26, 27, and 28,
$$\begin{aligned}&\mathrm{Cell}(a,\text{-},\mathrm{T}[id|\text{-},\text{-}]) \in \mathrm{Mem}(\sigma) \ \wedge\ \mathrm{Msg}(\mathrm{H},id,\mathrm{PurgeReq},a)_{\updownarrow} \in \mathrm{Cin}_{id}(\sigma)\\ \leadsto\ \ &\mathrm{Msg}(id,\mathrm{H},\mathrm{Purge},a) \in \mathrm{MinCout}_{id}(\sigma) \ \vee\ \mathrm{Msg}(id,\mathrm{H},\mathrm{Wb},a,\text{-}) \in \mathrm{MinCout}_{id}(\sigma)\\ \leadsto\ \ &\mathrm{Msg}(id,\mathrm{H},\mathrm{Purge},a) \in \mathrm{Min}(\sigma) \ \vee\ \mathrm{Msg}(id,\mathrm{H},\mathrm{Wb},a,\text{-}) \in \mathrm{Min}(\sigma)\\ \leadsto\ \ &\mathrm{Msg}(id,\mathrm{H},\mathrm{Purge},a)_{\uparrow} \in \mathrm{Min}(\sigma) \ \vee\ \mathrm{Msg}(id,\mathrm{H},\mathrm{Wb},a,\text{-})_{\uparrow} \in \mathrm{Min}(\sigma)\\ \leadsto\ \ &\mathrm{Cell}(a,\text{-},\mathrm{T}[dir_{-id},\text{-}]) \in \mathrm{Mem}(\sigma)\end{aligned}$$

Thus, $\mathrm{Cell}(a,\text{-},\mathrm{T}[id|\text{-},\text{-}]) \in \mathrm{Mem}(\sigma) \ \wedge\ \mathrm{Msg}(\mathrm{H},id,\mathrm{PurgeReq},a)_{\downarrow} \in \mathrm{Mout}(\sigma)$
$$\leadsto \quad \mathrm{Cell}(a,\text{-},\mathrm{T}[dir_{-id},\text{-}]) \in \mathrm{Mem}(\sigma)$$

This completes the proof according to Theorem-A. □

The proof above can be explained in a more intuitive way. The memory sends a PurgeReq message to cache *id* each time it changes the memory state to T[*id*|-,-]. When the PurgeReq message is received at cache *id*, if the cache state is Clean or Dirty, the cache sends a Purge or Wb message to the memory; otherwise the cache ignores the PurgeReq message. In the latter case, if the memory state remains as T[*id*|-,-], there must be a Purge or Wb message in transit from cache *id* to the memory according to Lemma 26 (note that the PurgeReq message has no following message since the memory cannot send any message to cache *id* while the memory state is T[*id*|-,-], and FIFO message passing guarantees that any preceding message must be received before the PurgeReq message is received). When the memory receives the Purge or Wb message, it removes the cache identifier from the directory.

Lemma 30 includes invariants about transient memory states. Invariant (1) ensures that the directory of a transient state will eventually become empty while each suspended writeback message remains unaffected. The proof is based on induction on the number of cache identifiers in the directory. It can be shown by checking each WP rule that suspended messages cannot be affected before the directory becomes empty. Invariant (2) ensures that a transient memory state will eventually become a stable memory state. The proof follows from the weak fairness of Rule MM11. This is critical to ensure that a stalled CacheReq message will be processed eventually.

Lemma 30 Given a WP sequence σ,

(1) $\text{Cell}(a,\text{-},\text{T}[\text{-},(id,\text{-})|\text{-}]) \in \text{Mem}(\sigma) \quad\leadsto\quad \text{Cell}(a,\text{-},\text{T}[\epsilon,(id,\text{-})|\text{-}]) \in \text{Mem}(\sigma)$

(2) $\text{Cell}(a,\text{-},\text{T}[\text{-},\text{-}]) \in \text{Mem}(\sigma) \quad\leadsto\quad \text{Cell}(a,\text{-},\text{C}[\text{-}]) \in \text{Mem}(\sigma)$

Lemma 31 ensures that any incoming message can become the first message regarding the address in the incoming queue (so that it can be processed by the corresponding protocol engine). This is a general form of Lemma 28.

Lemma 31 Given a WP sequence σ,

(1) $msg \in \text{Cin}_{id}(\sigma) \quad\leadsto\quad msg_{\uparrow} \in \text{Cin}_{id}(\sigma)$

(2) $msg \in \text{Min}(\sigma) \quad\leadsto\quad msg_{\uparrow} \in \text{Min}(\sigma)$

Lemma 32 ensures that if an address is cached in the CachePending state, the cache state will become Clean eventually. This is important to guarantee that a cache miss can be serviced in finite time.

Lemma 32 Given a WP sequence σ,

$\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \quad\leadsto\quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)$

Proof We first show some properties that are needed for the proof; all the properties can be verified by simply checking the WP rules.

- A cache sends a CacheReq message to the memory whenever it changes the state of a cache cell to CachePending. Note the CacheReq message has no following message when it is issued.

  $$\begin{aligned}&\mathrm{Cell}(a,\text{-},\mathrm{CachePending}) \notin \mathrm{Cache}_{id}(\sigma) \;\wedge\; \bigcirc\, \mathrm{Cell}(a,\text{-},\mathrm{CachePending}) \in \mathrm{Cache}_{id}(\sigma)\\ &\qquad\Rightarrow \quad \bigcirc\, \mathrm{Msg}(id,\mathrm{H},\mathrm{CacheReq},a)_{\downarrow} \in \mathrm{Cout}_{id}(\sigma)\end{aligned}$$

- The CachePending state can only be changed to the Clean state.

  $$\begin{aligned}&\mathrm{Cell}(a,\text{-},\mathrm{CachePending}) \in \mathrm{Cache}_{id}(\sigma) \;\wedge\; \bigcirc\, \mathrm{Cell}(a,\text{-},\mathrm{CachePending}) \notin \mathrm{Cache}_{id}(\sigma)\\ &\qquad\Rightarrow \quad \bigcirc\, \mathrm{Cell}(a,\text{-},\mathrm{Clean}) \in \mathrm{Cache}_{id}(\sigma)\end{aligned}$$

- A cache cannot send any message to the memory while the cache state is CachePending. Thus, a message that has no following message will remain as the last message.

  $$\begin{aligned}&\mathrm{Cell}(a,\text{-},\mathrm{CachePending}) \in \mathrm{Cache}_{id}(\sigma) \;\wedge\; msg_{\downarrow} \in \mathrm{MinCout}_{id}(\sigma)\\ &\qquad\Rightarrow \quad \bigcirc\, msg_{\downarrow} \in \mathrm{MinCout}_{id}(\sigma)\end{aligned}$$

We then prove the lemma under the assumption that the cache's outgoing queue contains a CacheReq message which has no following message.

- According to Theorem-C and Lemma 27,

  $$\begin{aligned}&\mathrm{Cell}(a,\text{-},\mathrm{CachePending}) \in \mathrm{Cache}_{id}(\sigma) \;\wedge\; \mathrm{Msg}(id,\mathrm{H},\mathrm{CacheReq},a)_{\downarrow} \in \mathrm{Cout}_{id}(\sigma)\\ \leadsto\quad &(\mathrm{Cell}(a,\text{-},\mathrm{CachePending}) \in \mathrm{Cache}_{id}(\sigma) \;\wedge\; \mathrm{Msg}(id,\mathrm{H},\mathrm{CacheReq},a)_{\downarrow} \in \mathrm{Min}(\sigma)) \;\vee\\ &\mathrm{Cell}(a,\text{-},\mathrm{Clean}) \in \mathrm{Cache}_{id}(\sigma)\end{aligned}$$

- According to Theorem-C and Lemma 31,

  $$\begin{aligned}&\mathrm{Cell}(a,\text{-},\mathrm{CachePending}) \in \mathrm{Cache}_{id}(\sigma) \;\wedge\; \mathrm{Msg}(id,\mathrm{H},\mathrm{CacheReq},a)_{\downarrow} \in \mathrm{Min}(\sigma)\\ \leadsto\quad &(\mathrm{Cell}(a,\text{-},\mathrm{CachePending}) \in \mathrm{Cache}_{id}(\sigma) \;\wedge\; \mathrm{Msg}(id,\mathrm{H},\mathrm{CacheReq},a)_{\updownarrow} \in \mathrm{Min}(\sigma)) \;\vee\\ &\mathrm{Cell}(a,\text{-},\mathrm{Clean}) \in \mathrm{Cache}_{id}(\sigma)\end{aligned}$$

- According to Lemmas 26, 27, and 31,

  $$\begin{aligned}&\mathrm{Cell}(a,\text{-},\mathrm{CachePending}) \in \mathrm{Cache}_{id}(\sigma) \;\wedge\; \mathrm{Msg}(id,\mathrm{H},\mathrm{CacheReq},a)_{\updownarrow} \in \mathrm{Min}(\sigma)\\ \leadsto\quad &\mathrm{Msg}(\mathrm{H},id,\mathrm{Cache},a,\text{-}) \in \mathrm{MoutCin}_{id}(\sigma)\\ \leadsto\quad &\mathrm{Msg}(\mathrm{H},id,\mathrm{Cache},a,\text{-}) \in \mathrm{Cin}_{id}(\sigma)\\ \leadsto\quad &\mathrm{Msg}(\mathrm{H},id,\mathrm{Cache},a,\text{-})_{\uparrow} \in \mathrm{Cin}_{id}(\sigma)\\ \leadsto\quad &\mathrm{Cell}(a,\text{-},\mathrm{Clean}) \in \mathrm{Cache}_{id}(\sigma)\end{aligned}$$

  $$\begin{aligned}\text{Thus,}\quad &\mathrm{Cell}(a,\text{-},\mathrm{CachePending}) \in \mathrm{Cache}_{id}(\sigma) \;\wedge\; \mathrm{Msg}(id,\mathrm{H},\mathrm{CacheReq},a)_{\downarrow} \in \mathrm{Cout}_{id}(\sigma)\\ &\quad\leadsto\quad \mathrm{Cell}(a,\text{-},\mathrm{Clean}) \in \mathrm{Cache}_{id}(\sigma)\end{aligned}$$

This completes the proof according to Theorem-A. □

The proof above can be explained in a more intuitive way. A cache generates a CacheReq message each time it changes the cache state of an address to CachePending. When the memory receives a CacheReq message from cache *id*, if the memory state is C[*dir*] where $id \notin dir$, the memory sends a Cache message to the cache; if the memory state is C[*dir*] or T[*dir*,*sm*] where $id \in dir$, the memory ignores the CacheReq message. In the latter case, if the cache state remains as CachePending, there must be a Cache message in transit from the memory to cache *id* according to Lemma 26 (note that the CacheReq message has no following message since the cache cannot issue any message while the cache state is CachePending, and FIFO message passing guarantees that any preceding message must be received before the CacheReq message is received). When the cache receives the Cache message, it caches the data and sets the cache state to Clean. It is worth pointing out that although the CacheReq message can be stalled at the memory, the stalled message will be processed eventually.

Lemma 33 ensures that if an address is cached in the WbPending state, the cache state will become Clean or the address will be purged from the cache. This is important to ensure that a writeback operation can always be completed.

Lemma 33 Given a WP sequence σ,

$$\text{Cell}(a,\text{-},\text{WbPending}) \in \text{Cache}_{id}(\sigma) \quad \rightsquigarrow \quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee\; a \notin \text{Cache}_{id}(\sigma)$$

Proof We first show that if a cache's outgoing queue has a Wb message, the cache state will become Clean or Invalid eventually.

$$\begin{array}{lll}
 & \text{Msg}(id,\text{H},\text{Wb},a,\text{-}) \in \text{Cout}_{id}(\sigma) & \\
\rightsquigarrow & \text{Msg}(id,\text{H},\text{Wb},a,\text{-}) \in \text{Min}(\sigma) & (\textit{Lemma 27}) \\
\rightsquigarrow & \text{Msg}(id,\text{H},\text{Wb},a,\text{-})_{\uparrow} \in \text{Min}(\sigma) & (\textit{Lemma 31}) \\
\rightsquigarrow & \text{Cell}(a,\text{-},\text{T}[\text{-},(id,\text{-})|\text{-}]) \in \text{Mem}(\sigma) & (\textit{Lemma 27}) \\
\rightsquigarrow & \text{Cell}(a,\text{-},\text{T}[\epsilon,(id,\text{-})|\text{-}]) \in \text{Mem}(\sigma) & (\textit{Lemma 30}) \\
\rightsquigarrow & \text{Msg}(\text{H},id,\text{WbAck},a) \in \text{Mout}(\sigma) \;\vee\; \text{Msg}(\text{H},id,\text{FlushAck},a) \in \text{Mout}(\sigma) & (\textit{Lemma 27}) \\
\rightsquigarrow & \text{Msg}(\text{H},id,\text{WbAck},a) \in \text{Cin}_{id}(\sigma) \;\vee\; \text{Msg}(\text{H},id,\text{FlushAck},a) \in \text{Cin}_{id}(\sigma) & (\textit{Lemma 27}) \\
\rightsquigarrow & \text{Msg}(\text{H},id,\text{WbAck},a)_{\uparrow} \in \text{Cin}_{id}(\sigma) \;\vee\; \text{Msg}(\text{H},id,\text{FlushAck},a)_{\uparrow} \in \text{Cin}_{id}(\sigma) & (\textit{Lemma 31}) \\
\rightsquigarrow & \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee\; a \notin \text{Cache}_{id}(\sigma) & (\textit{Lemma 27})
\end{array}$$

We then show that a cache sends a Wb message to the memory whenever it changes the state of a cache cell to WbPending. The following proposition can be verified by checking the WP rules.

$$\begin{array}{l}
\text{Cell}(a,\text{-},\text{WbPending}) \notin \text{Cache}_{id}(\sigma) \;\wedge\; \circ\, \text{Cell}(a,\text{-},\text{WbPending}) \in \text{Cache}_{id}(\sigma) \\
\qquad \Rightarrow \quad \circ\, \text{Msg}(id,\text{H},\text{Wb},a,\text{-}) \in \text{Cout}_{id}(\sigma)
\end{array}$$

This completes the proof according to Theorem-A. □

5.5.2 Liveness of WP

Lemma 34 ensures that whenever a processor intends to execute an instruction, the cache cell will be set to an appropriate state while the instruction remains in the processor-to-memory buffer. For a Loadl or Storel, the cache state will be set to Clean or Dirty; for a Commit, the cache state will be set to Clean or Invalid; for a Reconcile, the cache state will be set to Clean, Dirty or Invalid.

Lemma 34 Given a WP sequence σ,

(1) $\text{Loadl}(a) \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \text{Loadl}(a) \in \text{Pmb}_{id}(\sigma) \wedge$
$(\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma))$

(2) $\text{Storel}(a,\text{-}) \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \text{Storel}(a,\text{-}) \in \text{Pmb}_{id}(\sigma) \wedge$
$(\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma))$

(3) $\text{Commit}(a) \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \text{Commit}(a) \in \text{Pmb}_{id}(\sigma) \wedge$
$(\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee a \notin \text{Cache}_{id}(\sigma))$

(4) $\text{Reconcile}(a) \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \text{Reconcile}(a) \in \text{Pmb}_{id}(\sigma) \wedge$
$(\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \vee a \notin \text{Cache}_{id}(\sigma))$

Proof We first show that when a processor intends to execute an instruction, the cache cell will be set to an appropriate state. This can be represented by the following proposition; the proof follows from Lemmas 27, 32 and 33.

- $\text{Loadl}(a) \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma)$
- $\text{Storel}(a,\text{-}) \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma)$
- $\text{Commit}(a) \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee a \notin \text{Cache}_{id}(\sigma)$
- $\text{Reconcile}(a) \in \text{Pmb}_{id}(\sigma) \rightsquigarrow$
$\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \vee a \notin \text{Cache}_{id}(\sigma)$

We then show that an instruction can be completed only when the address is cached in an appropriate state. This can be represented by the following proposition, which can be verified by simply checking the WP rules that allow an instruction to be retired.

- $\langle t,\text{Loadl}(a)\rangle \in \text{Pmb}_{id}(\sigma) \wedge \circ \langle t,\text{Loadl}(a)\rangle \notin \text{Pmb}_{id}(\sigma)$
$\Rightarrow \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma)$
- $\langle t,\text{Storel}(a,\text{-})\rangle \in \text{Pmb}_{id}(\sigma) \wedge \circ \langle t,\text{Storel}(a,\text{-})\rangle \notin \text{Pmb}_{id}(\sigma)$
$\Rightarrow \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma)$
- $\langle t,\text{Commit}(a)\rangle \in \text{Pmb}_{id}(\sigma) \wedge \circ \langle t,\text{Commit}(a)\rangle \notin \text{Pmb}_{id}(\sigma)$
$\Rightarrow \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee a \notin \text{Cache}_{id}(\sigma)$
- $\langle t,\text{Reconcile}(a)\rangle \in \text{Pmb}_{id}(\sigma) \wedge \circ \langle t,\text{Reconcile}(a)\rangle \notin \text{Pmb}_{id}(\sigma)$
$\Rightarrow \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \vee a \notin \text{Cache}_{id}(\sigma)$

This completes the proof according to Theorem-B. □

The liveness of WP ensures that a memory instruction can always be completed eventually. This is described by the following theorem, which trivially follows from Lemma 34. Note that Rules P1, P2, P6, P7, P11, P15, P16, P17 and P20 are strongly fair.

Theorem 35 (Liveness of WP) Given a WP sequence σ,

(1) $\langle t,\text{Loadl(-)}\rangle \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \langle t,\text{-}\rangle \in \text{Mpb}_{id}(\sigma)$

(2) $\langle t,\text{Storel(-,-)}\rangle \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \langle t,\text{Ack}\rangle \in \text{Mpb}_{id}(\sigma)$

(3) $\langle t,\text{Commit(-)}\rangle \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \langle t,\text{Ack}\rangle \in \text{Mpb}_{id}(\sigma)$

(4) $\langle t,\text{Reconcile(-)}\rangle \in \text{Pmb}_{id}(\sigma) \rightsquigarrow \langle t,\text{Ack}\rangle \in \text{Mpb}_{id}(\sigma)$

M-engine Rules				
Msg from id	**Mstate**	**Action**	**Next Mstate**	
$\langle \text{CacheReq}, a \rangle$	$\text{Cell}(a,v,\text{C}[dir,hint])\ (id \notin dir)$	$\langle \text{Cache}, a, v \rangle \rightarrow id$	$\text{Cell}(a,v,\text{C}[id \mid dir,hint])$	SF
	$\text{Cell}(a,v,\text{C}[dir,hint])\ (id \in dir)$		$\text{Cell}(a,v,\text{C}[dir,hint])$	
	$\text{Cell}(a,v,\text{T}[dir,sm,hint])\ (id \notin dir)$	*stall message*	$\text{Cell}(a,v,\text{T}[dir,sm,hint])$	
	$\text{Cell}(a,v,\text{T}[dir,sm,hint])\ (id \in dir)$		$\text{Cell}(a,v,\text{T}[dir,sm,hint])$	
$\langle \text{Wb}, a, v \rangle$	$\text{Cell}(a,v_1,\text{C}[id \mid dir,hint])$	$\langle \text{PurgeReq}, a \rangle \rightarrow dir$	$\text{Cell}(a,v_1,\text{T}[dir,(id,v),hint])$	
	$\text{Cell}(a,v_1,\text{T}[id \mid dir,sm,hint])$		$\text{Cell}(a,v_1,\text{T}[dir,(id,v) \mid sm,hint])$	
$\langle \text{Purge}, a \rangle$	$\text{Cell}(a,v,\text{C}[id \mid dir,hint])$		$\text{Cell}(a,v,\text{C}[dir,hint])$	
	$\text{Cell}(a,v,\text{T}[id \mid dir,sm,hint])$		$\text{Cell}(a,v,\text{T}[dir,sm,id \mid hint])$	
	$\text{Cell}(a,\text{-},\text{T}[\epsilon,(id,v) \mid sm,hint])$	$\langle \text{FlushAck}, a \rangle \rightarrow id$	$\text{Cell}(a,v,\text{T}[\epsilon,sm,id \mid hint])$	
	$\text{Cell}(a,\text{-},\text{T}[\epsilon,(id,v),hint])$	$\langle \text{WbAck}, a \rangle \rightarrow id$	$\text{Cell}(a,v,\text{C}[id,hint])$	
	$\text{Cell}(a,v,\text{T}[\epsilon,\epsilon,hint])$		$\text{Cell}(a,v,\text{C}[\epsilon,hint])$	
	$\text{Cell}(a,v,\text{C}[dir,id \mid hint])\ (id \notin dir)$	$\langle \text{Cache}, a, v \rangle \rightarrow id$	$\text{Cell}(a,v,\text{C}[id \mid dir,hint])$	
	$\text{Cell}(a,v,\text{C}[dir,id \mid hint])$		$\text{Cell}(a,v,\text{C}[dir,hint])$	
	$\text{Cell}(a,v,\text{T}[dir,sm,id \mid hint])$		$\text{Cell}(a,v,\text{T}[dir,sm,hint])$	

Figure 5.10: Memory Engine Rules of an Update Protocol

5.6 An Update Protocol from WP

The WP protocol contains voluntary rules that can be invoked under various heuristic policies. Different heuristic policies can lead to different performance, but the soundness and liveness of the system are always guaranteed. To build a concrete adaptive protocol one has to decide only on the conditions in which each voluntary rule should be applied. This policy decision is taken on the basis of an expected or observed behavior of the program.

To demonstrate the use of voluntary rules, we build an update protocol from WP. The memory maintains some hueristic information about the cache sites that may need to be updated. It behaves as a hint for the invocation of the voluntary rule that sends a data copy to a cache site even though no cache request is received. The heuristic information is called soft state since it has no impact on the soundness and liveness of the protocol.

Figure 5.10 gives the M-engine rules for the update protocol (the processor and C-engine rules remain unchanged). When the memory receives a purge acknowledgment, it records the cache identifier as a hint for future cache update. Later when the memory is updated, it can send a data copy to the cache site in which the address was just purged. Note that a cache identifier in the heuristic information can be removed at any time without taking any action.

The correctness of the update protocol follows from two observations. First, with all the heuristic information removed, each rule of the update protocol can be projected to either a mandatory or voluntary rule of WP, or a rule that takes no action (that is, the rule has identical left-hand-side and right-hand-side). This guarantees the soundness of the protocol. Second, the action of each mandatory rule of WP can be invoked under the same condition in the update protocol, that is, its applicability is not contingent upon any heuristic information. This guarantees the liveness of the protocol.

Voluntary C-engine Rules

	Cstate	Action	Next Cstate
	Cell(a,-,Clean)	⟨Purge,a⟩ → H	$a \notin cache$
	Cell(a,v,Dirty)	⟨Wb,a,v⟩ → H	Cell(a,v,WbPending)
	$a \notin cache$	⟨CacheReq,a⟩ → H	Cell(a,-,CachePending)

Mandatory C-engine Rules

Msg from H	Cstate	Action	Next Cstate
⟨Cache,a,v⟩	$a \notin cache$		Cell(a,v,Clean)
	Cell(a,-,CachePending)		Cell(a,v,Clean)
⟨WbAck,a⟩	Cell(a,v,WbPending)		Cell(a,v,Clean)
⟨FlushAck,a⟩	Cell(a,-,WbPending)		$a \notin cache$
⟨PurgeReq,a⟩	Cell(a,-,Clean)	⟨Purge,a⟩ → H	$a \notin cache$
	Cell(a,v,Dirty)	⟨IsDirty,a⟩ → H	Cell(a,v,Dirty)
	Cell(a,v,WbPending)		Cell(a,v,WbPending)
	Cell(a,-,CachePending)		Cell(a,-,CachePending)
	$a \notin cache$		$a \notin cache$

Voluntary M-engine Rules

	Mstate	Action	Next Mstate
	Cell(a,v,C[dir]) ($id \notin dir$)	⟨Cache,a,v⟩ → id	Cell(a,v,C[$id \mid dir$])
	Cell(a,v,C[dir]) ($dir \neq \epsilon$)	⟨PurgeReq,a⟩ → dir	Cell(a,v,T[dir,ϵ,ϵ])

Mandatory M-engine Rules

Msg from id	Mstate	Action	Next Mstate	
⟨CacheReq,a⟩	Cell(a,v,C[dir]) ($id \notin dir$)	⟨Cache,a,v⟩ → id	Cell(a,v,C[$id \mid dir$])	SF
	Cell(a,v,C[dir]) ($id \in dir$)		Cell(a,v,C[dir])	
	Cell(a,v,T[dir,sm,dir_1]) ($id \notin dir$)	*stall message*	Cell(a,v,T[dir,sm,dir_1])	
	Cell(a,v,T[dir,sm,dir_1]) ($id \in dir$)		Cell(a,v,T[dir,sm,dir_1])	
⟨Wb,a,v⟩	Cell(a,v_1,C[$id \mid dir$])	⟨PurgeReq,a⟩ → dir	Cell(a,v_1,T[dir,(id,v),ϵ])	
	Cell(a,v_1,T[$id \mid dir$,sm,dir_1])		Cell(a,v_1,T[dir,(id,v)$\mid sm$,dir_1])	
	Cell(a,v_1,T[dir,sm,$id \mid dir_1$])		Cell(a,v_1,T[dir,(id,v)$\mid sm$,dir_1])	
⟨IsDirty,a⟩	Cell(a,v,T[$id \mid dir$,sm,dir_1])		Cell(a,v,T[dir,sm,$id \mid dir_1$])	
⟨Purge,a⟩	Cell(a,v,C[$id \mid dir$])		Cell(a,v,C[dir])	
	Cell(a,v,T[$id \mid dir$,sm,dir_1])		Cell(a,v,T[dir,sm,dir_1])	
	Cell(a,-,T[ϵ,(id,v)$\mid sm$,dir_1])	⟨FlushAck,a⟩ → id	Cell(a,v,T[ϵ,sm,dir_1])	
	Cell(a,-,T[ϵ,(id,v),dir_1])	⟨WbAck,a⟩ → id	Cell(a,v,C[$id \mid dir_1$])	
	Cell(a,v,T[ϵ,ϵ,dir_1])		Cell(a,v,C[dir_1])	

Figure 5.11: An Alternative Writer-Push Protocol

5.7 An Alternative Writer-Push Protocol

The design of sophisticated coherence protocols involves many design choices. Different choices represent different tradeoffs that have different implications on performance and implementation complexity. In WP, for example, when a cache receives a purge request on a dirty cell, it forces the data to be written back to the memory. This allows the memory to maintain only one directory that records the cache sites whose copies will be purged or written back. Another option is to allow the cache to keep its dirty copy as long as it notifies the memory that the data has been modified. This avoids unnecessary writeback operations, but the memory may need to maintain more information.

Figure 5.11 gives the cache engine and memory engine rules of an alternative writer-push protocol (the processor rules remain the same as those in the original WP protocol). A new message IsDirty is introduced. The T[dir,sm,dir_1] state means that the memory has sent purge requests to cache sites dir, and has received IsDirty messages from cache sites dir_1.

Chapter 6

The Migratory Cache Coherence Protocol

When a memory location is accessed predominantly by one processor, all the operations performed at that site should be inexpensive. The Migratory protocol is suitable for this situation. It allows each address to be cached in at most one cache so that both commit and reconcile operations can complete regardless of whether the data has been modified or not. Consequently, memory accesses from another site can be expensive since the exclusive copy must be migrated to that site before the accesses can be performed.

Section 6.1 describes the cache and memory states and protocol messages of Migratory. We present the imperative rules of Migratory in Section 6.2, and give the complete protocol in Section 6.3. The soundness and liveness of Migratory are proved in Sections 6.4 and 6.5, respectively.

6.1 The System Configuration of the Migratory Protocol

Figure 6.1 defines the system configuration of the Migratory protocol. The Migratory protocol employs two stable cache states, Clean and Dirty, and one transient cache state, CachePending, which implies that the address is uncached and a cache request has been sent to the memory. Each memory cell maintains a memory state, which can be C[ϵ], C[id] or T[id]. The C[ϵ] state means that the address is currently uncached in any cache. Both the C[id] and T[id] states imply that the address is cached exclusively in cache site id; the distinction between them is that T[id] also implies that a flush request has been sent to the cache site. As will be seen, CachePending and T[id] are introduced purely for liveness reason, and are not used in the imperative rules.

In Migratory, there are three imperative messages, Cache, Purge and Flush, and two directive messages, CacheReq and FlushReq. The informal meaning of each message is as follows:

- Cache: the memory supplies a data copy to the cache.
- FlushReq: the memory requests the cache to flush its cache copy.

SYS	$\equiv$	Sys(MSITE, SITEs)	*System*
MSITE	$\equiv$	Msite(MEM, IN, OUT)	*Memory Site*
MEM	$\equiv$	ϵ ‖ Cell(a,v,MSTATE) | MEM	*Memory*
SITEs	$\equiv$	SITE ‖ SITE | SITEs	*Set of Cache Sites*
SITE	$\equiv$	Site(id, CACHE, IN, OUT, PMB, MPB, PROC)	*Cache Site*
CACHE	$\equiv$	ϵ ‖ Cell(a,v,CSTATE) | CACHE	*Cache*
IN	$\equiv$	ϵ ‖ MSG $\odot$ IN	*Incoming Queue*
OUT	$\equiv$	ϵ ‖ MSG $\otimes$ OUT	*Outgoing Queue*
MSG	$\equiv$	Msg(src,$dest$,CMD,a,v)	*Message*
MSTATE	$\equiv$	C[ϵ] ‖ C[id] ‖ T[id]	*Memory state*
CSTATE	$\equiv$	Clean ‖ Dirty ‖ CachePending	*Cache State*
CMD	$\equiv$	Cache ‖ Purge ‖ Flush ‖	
		CacheReq ‖ FlushReq	*Command*

Figure 6.1: System Configuration of Migratory

- Purge: the cache informs the memory that its cache copy has been purged.
- Flush: the cache sends the dirty data back to the memory.
- CacheReq: the cache requests a data copy from the memory.

6.2 The Imperative Rules of the Migratory Protocol

We develop a set of imperative rules that determine the soundness of the system. The imperative rules include the processor rules, the cache engine rules and the memory engine rules.

Processor Rules: The imperative processor rules of Migratory contain all the imperative processor rules of Base (see Section 4.3). In addition, the *Commit-on-Dirty* rule allows a Commit instruction to complete even when the address is cached in the Dirty state, and the *Reconcile-on-Clean* rule allows a Reconcile instruction to complete even when the address is cached in the Clean state.

Commit-on-Dirty Rule

Site(id, Cell(a,v,Dirty) | $cache$, in, out, $\langle t$,Commit(a)$\rangle$;pmb, mpb, $proc$)
$\rightarrow$ Site(id, Cell(a,v,Dirty) | $cache$, in, out, pmb, $mpb|\langle t$,Ack$\rangle$, $proc$)

Reconcile-on-Clean Rule

Site(id, Cell(a,v,Clean) | $cache$, in, out, $\langle t$,Reconcile(a)$\rangle$;pmb, mpb, $proc$)
$\rightarrow$ Site(id, Cell(a,v,Clean) | $cache$, in, out, pmb, $mpb|\langle t$,Ack$\rangle$, $proc$)

C-engine Rules: A cache can purge a clean copy and inform the memory via a Purge message. It can also flush a dirty copy and write the data back to the memory via a Flush message. It is worth noting that no acknowledgment is needed from the memory for the flush operation.

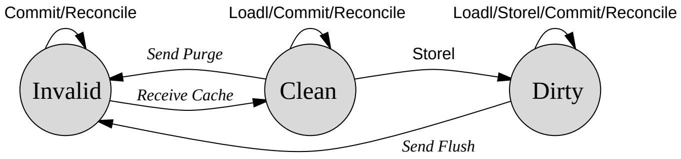


Figure 6.2: Cache State Transitions of Migratory's Imperative Operations

When a cache receives a Cache message from the memory, it caches the data in the Clean state. Figure 6.2 shows the cache state transitions due to imperative operations.

C-Send-Purge Rule

$\quad$ Site(id, Cell(a,-,Clean) | $cache$, in, out, pmb, mpb, $proc$)

$\to$ Site(id, $cache$, in, $out \otimes$ Msg(id,H,Purge,a), pmb, mpb, $proc$)

C-Send-Flush Rule

$\quad$ Site(id, Cell(a,v,Dirty) | $cache$, in, out, pmb, mpb, $proc$)

$\to$ Site(id, $cache$, in, $out \otimes$ Msg(id,H,Flush,a,v), pmb, mpb, $proc$)

C-Receive-Cache Rule

$\quad$ Site(id, $cache$, Msg(H,id,Cache,a,v) $\odot$ in , out, pmb, mpb, $proc$)

$\to$ Site(id, Cell(a,v,Clean) | $cache$, in, out, pmb, mpb, $proc$)

M-engine Rules The memory can send a Cache message to supply a data copy to a cache, if the address is currently not cached in any cache. When the memory receives a Purge message, it removes the cache identifier from the memory state. When the memory receives a Flush message, it updates the memory with the flushed data and changes the memory state accordingly.

M-Send-Cache Rule

$\quad$ Msite(Cell(a,v,C[ϵ]) | mem, in, out)

$\to$ Msite(Cell(a,v,C[id]) | mem, in, $out \otimes$ Msg(H,id,Cache,a,v))

M-Receive-Purge Rule

$\quad$ Msite(Cell(a,v,C[id]) | mem, Msg(id,H,Purge,a) $\odot$ in , out)

$\to$ Msite(Cell(a,v,C[ϵ]) | mem, in, out)

M-Receive-Flush Rule

$\quad$ Msite(Cell(a,-,C[id]) | mem, Msg(id,H,Flush,a,v) $\odot$ in , out)

$\to$ Msite(Cell(a,v,C[ϵ]) | mem, in, out)

Figure 6.3 summarizes the imperative rules of Migratory. When an instruction is retired, it is removed from the processor-to-memory buffer while the corresponding response is supplied to the memory-to-processor buffer. When a message is received, it is removed from the incoming queue.

Imperative Processor Rules				
Instruction	**Cstate**	**Action**	**Next Cstate**	
Loadl(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	IP1
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	IP2
Storel(a,v)	Cell(a,-,Clean)	*retire*	Cell(a,v,Dirty)	IP3
	Cell(a,-,Dirty)	*retire*	Cell(a,v,Dirty)	IP4
Commit(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	IP5
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	IP6
	$a \notin cache$	*retire*	$a \notin cache$	IP7
Reconcile(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	IP8
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	IP9
	$a \notin cache$	*retire*	$a \notin cache$	IP10

Imperative C-engine Rules				
Msg from H	**Cstate**	**Action**	**Next Cstate**	
	Cell(a,-,Clean)	$\langle$Purge,$a\rangle \rightarrow$ H	$a \notin cache$	IC1
	Cell(a,v,Dirty)	$\langle$Flush,a,$v\rangle \rightarrow$ H	$a \notin cache$	IC2
$\langle$Cache,a,$v\rangle$	$a \notin cache$		Cell(a,v,Clean)	IC3

Imperative M-engine Rules				
Msg from id	**Mstate**	**Action**	**Next Mstate**	
	Cell(a,v,C[ϵ])	$\langle$Cache,a,$v\rangle \rightarrow id$	Cell(a,v,C[id])	IM1
$\langle$Purge,$a\rangle$	Cell(a,v,C[id])		Cell(a,v,C[ϵ])	IM2
$\langle$Flush,a,$v\rangle$	Cell(a,-,C[id])		Cell(a,v,C[ϵ])	IM3

Figure 6.3: Imperative Rules of Migratory

6.3 The Migratory Protocol

To ensure liveness, we introduce two directive messages, CacheReq and FlushReq. Whenever necessary, a cache can send a CacheReq message to request a data copy from the memory, and the memory can send a FlushReq message to force a cache copy to be flushed. In addition, we augment certain cache and memory states with information regarding outstanding directive messages. The Invalid cache state in the imperative rules becomes Invalid or CachePending, depending on whether a cache request has been sent to the memory. At the memory side, the C[id] state in the imperative rules becomes C[id] or T[id], depending on whether a flush request has been sent to the cache site.

Figure 6.4 gives the rules of the Migratory protocol. The tabular description can be easily translated into formal TRS rules. A mandatory rule marked with 'SF' requires strong fairness to ensure that each memory instruction can be retired eventually. A retired instruction is immediately removed from the processor-to-memory buffer, while a stalled instruction remains for later processing. When a message is processed, it is removed from the incoming queue; when a message is stalled, it remains in the incoming queue but does not block following messages. The Migratory protocol assumes FIFO message passing for protocol messages with the same address.

For each address, the memory maintains which site currently has cached the address. When it receives a cache request while the address is cached in another cache, it stalls the cache request and sends a flush request to the cache to force it to flush its copy. Note that there can be multiple stalled cache requests regarding the same address. The strong fairness of Rule MM1

Mandatory Processor Rules

Instruction	**Cstate**	**Action**	**Next Cstate**		
Loadl(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	P1	SF
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	P2	SF
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P3	
	$a \notin$ *cache*	*stall*, ⟨CacheReq,a⟩ → H	Cell(a,-,CachePending)	P4	
Storel(a,v)	Cell(a,-,Clean)	*retire*	Cell(a,v,Dirty)	P5	SF
	Cell(a,-,Dirty)	*retire*	Cell(a,v,Dirty)	P6	SF
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P7	
	$a \notin$ *cache*	*stall*, ⟨CacheReq,a⟩ → H	Cell(a,-,CachePending)	P8	
Commit(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	P9	SF
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	P10	SF
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P11	
	$a \notin$ *cache*	*retire*	$a \notin$ *cache*	P12	SF
Reconcile(a)	Cell(a,v,Clean)	*retire*	Cell(a,v,Clean)	P13	SF
	Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)	P14	SF
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P15	
	$a \notin$ *cache*	*retire*	$a \notin$ *cache*	P16	SF

Voluntary C-engine Rules

	Cstate	**Action**	**Next Cstate**	
	Cell(a,-,Clean)	⟨Purge,a⟩ → H	$a \notin$ *cache*	VC1
	Cell(a,v,Dirty)	⟨Flush,a,v⟩ → H	$a \notin$ *cache*	VC2
	$a \notin$ *cache*	⟨CacheReq,a⟩ → H	Cell(a,-,CachePending)	VC3

Mandatory C-engine Rules

Msg from H	**Cstate**	**Action**	**Next Cstate**	
⟨Cache,a,v⟩	$a \notin$ *cache*		Cell(a,v,Clean)	MC1
	Cell(a,-,CachePending)		Cell(a,v,Clean)	MC2
⟨FlushReq,a⟩	Cell(a,-,Clean)	⟨Purge,a⟩ → H	$a \notin$ *cache*	MC3
	Cell(a,v,Dirty)	⟨Flush,a,v⟩ → H	$a \notin$ *cache*	MC4
	Cell(a,-,CachePending)		Cell(a,-,CachePending)	MC5
	$a \notin$ *cache*		$a \notin$ *cache*	MC6

Voluntary M-engine Rules

	Mstate	**Action**	**Next Mstate**	
	Cell(a,v,C[ϵ])	⟨Cache,a,v⟩ → id	Cell(a,v,C[id])	VM1
	Cell(a,v,C[id])	⟨FlushReq,a⟩ → id	Cell(a,v,T[id])	VM2

Mandatory M-engine Rules

Msg from id	**Mstate**	**Action**	**Next Mstate**		
⟨CacheReq,a⟩	Cell(a,v,C[ϵ])	⟨Cache,a,v⟩ → id	Cell(a,v,C[id])	MM1	SF
	Cell(a,v,C[id_1]) ($id_1 \neq id$)	*stall message* ⟨FlushReq,a⟩ → id_1	Cell(a,v,T[id_1])	MM2	
	Cell(a,v,C[id])		Cell(a,v,C[id])	MM3	
	Cell(a,-,T[id_1]) ($id_1 \neq id$)	*stall message*	Cell(a,-,T[id_1])	MM4	
	Cell(a,-,T[id])		Cell(a,-,T[id])	MM5	
⟨Purge,a⟩	Cell(a,v,C[id])		Cell(a,v,C[ϵ])	MM6	
	Cell(a,v,T[id])		Cell(a,v,C[ϵ])	MM7	
⟨Flush,a,v⟩	Cell(a,-,C[id])		Cell(a,v,C[ϵ])	MM8	
	Cell(a,-,T[id])		Cell(a,v,C[ϵ])	MM9	

Figure 6.4: The Migratory Protocol

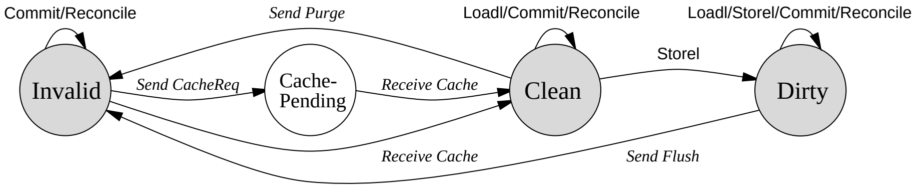


Figure 6.5: Cache State Transitions of Migratory

ensures that a cache request cannot be blocked forever while cache requests from other sites are serviced repeatedly. The Migratory protocol assumes proper buffer management that can be characterized as follows:

Migratory's buffer management:

$$\begin{array}{lll} msg_1 \odot msg_2 & \equiv & msg_2 \odot msg_1 \\ \quad \mathit{if} & & (\mathrm{Cmd}(msg_1) = \mathrm{CacheReq} \ \vee \ \mathrm{Cmd}(msg_2) = \mathrm{CacheReq}) \ \wedge \\ & & (\mathrm{Src}(msg_1) \neq \mathrm{Src}(msg_2) \ \vee \ \mathrm{Addr}(msg_1) \neq \mathrm{Addr}(msg_2)) \end{array}$$

Voluntary Rules: At any time, a cache can purge a clean copy and notify the memory of the purge operation via a Purge message. It can also flush a dirty copy and write the data back to the memory via a Flush message. A cache can send a cache request to the memory to request an exclusive copy for an uncached address. Figure 6.5 shows the cache state transitions of Migratory.

On the other hand, the memory can voluntarily send an exclusive copy to a cache, if the address is currently not cached in any cache. If the memory state shows that an address is cached in some cache, the memory can voluntarily send a flush request to the cache to force the data to be flushed from the cache.

The voluntary rules allow the memory to supply a data copy without a request from the cache, and a cache to flush a data copy without a request from the memory. This can cause unexpected situations if a request is received after the requested action has been performed.

- Simultaneous Cache and CacheReq. Suppose initially the address is not cached in any cache. The memory sends a Cache message to a cache, while the cache sends a CacheReq to the memory. The CacheReq message will be discarded when it is received at the memory (Rules MM3 & MM5).
- Simultaneous Purge and FlushReq. Suppose initially a clean copy of the address is cached in a cache site. The cache purges the clean copy and sends a Purge message to the memory, while the memory sends a FlushReq message to the cache. The FlushReq message will be discarded when it received at the cache (Rules MC5 and MC6).
- Simultaneous Flush and FlushReq. Suppose initially a dirty copy of the address is cached in a cache site. The cache flushes the dirty copy and sends a Flush message to the memory,

while the memory sends a FlushReq message to the cache. The FlushReq message will be discarded when it received at the cache (Rules MC5 and MC6).

Optimizations: In Migratory, an instruction is always stalled when the address is cached in a transient state. A potential optimization is to allow a Commit or Reconcile instruction to be completed regardless of the current cache state. This cannot compromise the soundness of the protocol, since no distinction is drawn between CachePending and Invalid in terms of soundness.

FIFO Message Passing: The liveness of Migratory is contingent upon FIFO message passing. Consider a scenario in which a cache sends a Flush message to write a data copy back to the memory, followed by a CacheReq message to request a data copy from the memory. The CacheReq message would be discarded if it were received before the Flush message, which could lead to a deadlock or livelock situation. The Migratory protocol requires that FIFO ordering be maintained in the following cases:

- Cache followed by FlushReq. The memory sends a Cache message to a cache to supply a data copy, and then sends a FlushReq message to flush the copy.
- Purge or Flush followed by CacheReq. A cache sends a Purge or Flush message to the memory, and then sends a CacheReq message to request a data copy from the memory.

It is worth mentioning that there are also cases that require no FIFO message passing although multiple messages regarding the same address are involved. This happens when the preceding message is a directive message.

- CacheReq followed by Purge. A cache sends a CacheReq message to the memory. Before the CacheReq message is received, the memory voluntarily sends a Cache message to the cache. The cache receives the Cache message and caches the data in the Clean state. The cache then purges the clean copy and sends a Purge message to the memory. The CacheReq message and the Purge message can be reordered.
- CacheReq followed by Flush. A cache sends a CacheReq message to the memory. Before the CacheReq message is received, the memory voluntarily sends a Cache message to the cache. The cache receives the Cache message and caches the data in the Clean state. The processor performs a Storel instruction, and the cache state becomes Dirty. The cache then flushes the dirty copy and sends a Flush message to the memory. The CacheReq message and the Flush message can be reordered.
- FlushReq followed by Cache. The memory sends a FlushReq message to the cache where the address is cached. Before the FlushReq message is received, the cache voluntarily flushes the cache copy and sends a Purge or Flush message to the memory. The memory receives the message, and then sends a Cache message to the cache. The FlushReq message and the Cache message can be reordered.

6.4 Soundness Proof of the Migratory Protocol

In this section, we prove the soundness of Migratory by showing that CRF can simulate Migratory. We define a mapping function from Migratory to CRF, and show that any imperative rule of Migratory can be simulated in CRF with respect to the mapping function. The soundness of Migratory follows from the fact that the integrated rules can be derived from the imperative and directive rules, and the directive rules cannot affect the soundness of the system. We first present some invariants that will be used throughout the proof; all the invariants are given with respect to the imperative rules.

6.4.1 Some Invariants of Migratory

Lemma 36 describes the correspondence between memory states, cache states and messages in transit. If the memory state shows that an address is cached in a cache, then the address must be cached in the cache, or a Cache, Purge or Flush message regarding the address is in transit between the memory and the cache. On the other hand, if an address is cached in a cache and the cache state is Clean or Dirty, or if a Cache, Purge or Flush message is in transit between the memory and the cache, then the cache identifier must appear in the corresponding memory state. Furthermore, a clean cache cell always contains the same value as the memory cell.

Lemma 36 Given a Migratory term s,

$$\begin{array}{l} \text{Cell}(a,v,\text{C}[id]) \in \text{Mem}(s) \quad \Leftrightarrow \\ \qquad \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(s) \;\vee\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(s) \;\vee \\ \qquad \text{Msg}(\text{H},id,\text{Cache},a,v) \in \text{MoutCin}_{id}(s) \;\vee\; \text{Msg}(id,\text{H},\text{Purge},a) \in \text{MinCout}_{id}(s) \;\vee \\ \qquad \text{Msg}(id,\text{H},\text{Flush},a,\text{-}) \in \text{MinCout}_{id}(s) \end{array}$$

Proof The proof is based on induction on rewriting steps. The invariant holds trivially for the initial term where all caches and queues are empty. It can be shown by checking each rule that, if the invariant holds for a term, then it still holds after the term is rewritten according to that rule. □

6.4.2 Mapping from Migratory to CRF

We define a mapping function that maps terms of Migratory to terms of CRF. The mapping function is based on the notion of drained terms, in which all caches and message queues are empty. We force all the cache cells to be flushed to ensure that the memory in a drained term always contains the most up-to-date data. This can be achieved by applying the purge rule on each clean cache cell and the flush rule on each dirty cache cell. In addition, we use forward draining to drain protocol messages from network queues.

Definition 37 (Draining Rules) Given a Migratory term s, dr(s) is the normal form of s with respect to the following draining rules:

$$\begin{array}{lll} D & \equiv & \{ \;\; \textit{C-Send-Purge}, \textit{C-Send-Flush}, \textit{C-Receive-Cache}, \textit{M-Receive-Purge}, \\ & & \;\;\;\;\; \textit{M-Receive-Flush}, \textit{Message-Cache-to-Mem}, \textit{Message-Mem-to-Cache} \;\; \} \end{array}$$

Lemma 38 D is strongly terminating and confluent, that is, rewriting a Migratory term with respect to the draining rules always terminates and reaches the same normal form, regardless of the order in which the rules are applied.

Proof The termination is obvious because throughout the draining process, only one new message (Purge or Flush) can be generated for each cache cell, and each message is consumed when it is received. The confluence follows from the fact that the draining rules do not interfere with each other. □

Lemma 39 is obvious since the draining rules cannot modify the processor-to-memory buffers, memory-to-processor buffers and processors. Lemma 40 ensures that all the caches and message queues are empty in a drained term. The proof follows from Lemma 36, which guarantees that all the messages can be consumed when they are received.

Lemma 39 Given a Migratory term s,

(1) $\mathrm{Pmb}_{id}(s) \quad = \quad \mathrm{Pmb}_{id}(\mathrm{dr}(s))$

(2) $\mathrm{Mpb}_{id}(s) \quad = \quad \mathrm{Mpb}_{id}(\mathrm{dr}(s))$

(3) $\mathrm{Proc}_{id}(s) \quad = \quad \mathrm{Proc}_{id}(\mathrm{dr}(s))$

Lemma 40 Given a Migratory term s,

(1) $\mathrm{Cache}_{id}(\mathrm{dr}(s)) \quad = \quad \epsilon$

(2) $\mathrm{Min}(\mathrm{dr}(s)) \quad = \quad \epsilon$

(3) $\mathrm{Mout}(\mathrm{dr}(s)) \quad = \quad \epsilon$

(4) $\mathrm{Cin}_{id}(\mathrm{dr}(s)) \quad = \quad \epsilon$

(5) $\mathrm{Cout}_{id}(\mathrm{dr}(s)) \quad = \quad \epsilon$

Lemma 41 ensures that, given a Migratory term, if an address is cached in some cache, then the memory cell of the address in the corresponding drained term contains the same value. The proof simply follows from Lemma 36 and the confluence property of the draining rules.

Lemma 41 Given a Migratory term s,

(1) $\mathrm{Cell}(a,v,\mathrm{Clean}) \in \mathrm{Cache}_{id}(s) \quad \Rightarrow \quad \mathrm{Cell}(a,v,\mathrm{C}[\epsilon]) \in \mathrm{Mem}(\mathrm{dr}(s))$

(2) $\mathrm{Cell}(a,v,\mathrm{Dirty}) \in \mathrm{Cache}_{id}(s) \quad \Rightarrow \quad \mathrm{Cell}(a,v,\mathrm{C}[\epsilon]) \in \mathrm{Mem}(\mathrm{dr}(s))$

Definition 42 (Mapping from Migratory to CRF) Given a Migratory term s, the corresponding CRF term $\mathrm{f}(s)$ is the drained term $\mathrm{dr}(s)$ with all message queues and cache identifiers removed.

6.4.3 Simulation of Migratory in CRF

Theorem 43 (CRF Simulates Migratory) Given Migratory terms s_1 and s_2,

$$s_1 \rightarrow s_2 \quad \text{in Migratory} \qquad \Rightarrow \qquad \mathrm{f}(s_1) \twoheadrightarrow \mathrm{f}(s_2) \quad \text{in CRF}$$

Proof The proof is based on a case analysis on the imperative rule used in the rewriting of '$s_1 \to s_2$' in Migratory. Let a be the address and id the cache identifier.

Imperative Processor Rules

- If Rule IP1 (*Loadl-on-Clean*) applies, then

$$\begin{array}{lll} & \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(s_1) \;\wedge\; \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(s_2) & \\ \Rightarrow & \text{Cell}(a,v,\text{C}[\epsilon]) \in \text{Mem}(s_1) \;\wedge\; \text{Cell}(a,v,\text{C}[\epsilon]) \in \text{Mem}(s_2) & (\textit{Lemma 41}) \\ \Rightarrow & \text{f}(s_1) \twoheadrightarrow \text{f}(s_2) \qquad (\textit{CRF-Cache, CRF-Loadl \& CRF-Purge}) & \end{array}$$

- If Rule IP2 (*Loadl-on-Dirty*) applies, then

$$\begin{array}{lll} & \text{Cell}(a,v,\text{Dirty}) \in \text{Cache}_{id}(s_1) \;\wedge\; \text{Cell}(a,v,\text{Dirty}) \in \text{Cache}_{id}(s_2) & \\ \Rightarrow & \text{Cell}(a,v,\text{C}[\epsilon]) \in \text{Mem}(s_1) \;\wedge\; \text{Cell}(a,v,\text{C}[\epsilon]) \in \text{Mem}(s_2) & (\textit{Lemma 41}) \\ \Rightarrow & \text{f}(s_1) \twoheadrightarrow \text{f}(s_2) \qquad (\textit{CRF-Cache, CRF-Loadl \& CRF-Purge}) & \end{array}$$

- If Rule IP3 (*Storel-on-Clean*) applies, then

$$\begin{array}{lll} & \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(s_1) \;\wedge\; \text{Cell}(a,v,\text{Dirty}) \in \text{Cache}_{id}(s_2) & \\ \Rightarrow & \text{Cell}(a,\text{-},\text{C}[\epsilon]) \in \text{Mem}(s_1) \;\wedge\; \text{Cell}(a,v,\text{C}[\epsilon]) \in \text{Mem}(s_2) & (\textit{Lemma 41}) \\ \Rightarrow & \text{f}(s_1) \twoheadrightarrow \text{f}(s_2) \qquad (\textit{CRF-Cache, CRF-Storel, CRF-Writeback \& CRF-Purge}) & \end{array}$$

- If Rule IP4 (*Storel-on-Dirty*) applies, then

$$\begin{array}{lll} & \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(s_1) \;\wedge\; \text{Cell}(a,v,\text{Dirty}) \in \text{Cache}_{id}(s_2) & \\ \Rightarrow & \text{Cell}(a,\text{-},\text{C}[\epsilon]) \in \text{Mem}(s_1) \;\wedge\; \text{Cell}(a,v,\text{C}[\epsilon]) \in \text{Mem}(s_2) & (\textit{Lemma 41}) \\ \Rightarrow & \text{f}(s_1) \twoheadrightarrow \text{f}(s_2) \qquad (\textit{CRF-Cache, CRF-Storel, CRF-Writeback \& CRF-Purge}) & \end{array}$$

- If Rule IP5 (*Commit-on-Clean*), IP6 (*Commit-on-Dirty*) or IP7 (*Commit-on-Invalid*) applies, then

$$\text{f}(s_1) \to \text{f}(s_2) \qquad (\textit{CRF-Commit})$$

- If Rule IP8 (*Reconcile-on-Clean*), IP9 (*Reconcile-on-Dirty*) or IP10 (*Reconcile-on-Invalid*) applies, then

$$\text{f}(s_1) \to \text{f}(s_2) \qquad (\textit{CRF-Reconcile})$$

Imperative M-engine Rules

- If Rule IM1 (*M-Send-Cache*) applies, then

$$\text{f}(s_1) = \text{f}(s_2)$$

Draining Rules

- If Rule IC1 (*C-Send-Purge*), IC2 (*C-Send-Flush*), IC3 (*C-Receive-Cache*), IM1 (*M-Receive-Purge*), IM2 (*M-Receive-Flush*), *Message-Cache-to-Mem* or *Message-Mem-to-Cache* applies, then

$$\text{f}(s_1) = \text{f}(s_2) \qquad (\textit{Since the rule or its backward version is a draining rule})$$

Figure 6.6 summarizes the simulation proof. □

Migratory Imperative Rule	**CRF Rules**
IP1 (*Loadl-on-Clean*)	*CRF-Cache* + *CRF-Loadl* + *CRF-Purge*
IP2 (*Loadl-on-Dirty*)	*CRF-Cache* + *CRF-Loadl* + *CRF-Purge*
IP3 (*Storel-on-Clean*)	*CRF-Cache* + *CRF-Storel* + *CRF-Writeback* + *CRF-Purge*
IP4 (*Storel-on-Dirty*)	*CRF-Cache* + *CRF-Storel* + *CRF-Writeback* + *CRF-Purge*
IP5 (*Commit-on-Clean*)	*CRF-Commit*
IP6 (*Commit-on-Dirty*)	*CRF-Commit*
IP7 (*Commit-on-Invalid*)	*CRF-Commit*
IP8 (*Reconcile-on-Clean*)	*CRF-Reconcile*
IP9 (*Reconcile-on-Dirty*)	*CRF-Reconcile*
IP10 (*Reconcile-on-Invalid*)	*CRF-Reconcile*
IC1 (*C-Send-Purge*)	ϵ
IC2 (*C-Send-Flush*)	ϵ
IC3 (*C-Receive-Cache*)	ϵ
IM1 (*M-Send-Cache*)	ϵ
IM2 (*M-Receive-Purge*)	ϵ
IM3 (*M-Receive-Flush*)	ϵ
Message-Mem-to-Cache	ϵ
Message-Cache-to-Mem	ϵ

Figure 6.6: Simulation of Migratory in CRF

6.4.4 Soundness of Migratory

The Migratory protocol defined in Figure 6.4 contains integrated rules that are derived from the imperative and directive rules of Migratory. The imperative rules are given in Section 6.2. There are four directive rules that can be used to generate or discard directive messages.

C-Send-CacheReq Rule

$\quad\text{Site}(id,\ cache,\ in,\ out,\ pmb,\ mpb,\ proc)$

$\rightarrow\ \text{Site}(id,\ cache,\ in,\ out \otimes \text{Msg}(id,\text{H},\text{CacheReq},a),\ pmb,\ mpb,\ proc)$

C-Receive-FlushReq Rule

$\quad\text{Site}(id,\ cache,\ \text{Msg}(\text{H},id,\text{FlushReq},a) \odot in,\ out,\ pmb,\ mpb,\ proc)$

$\rightarrow\ \text{Site}(id,\ cache,\ in,\ out,\ pmb,\ mpb,\ proc)$

M-Send-FlushReq Rule

$\quad\text{Msite}(mem,\ in,\ out)$

$\rightarrow\ \text{Msite}(mem,\ in,\ out \otimes \text{Msg}(\text{H},id,\text{FlushReq},a))$

M-Receive-CacheReq Rule

$\quad\text{Msite}(mem,\ \text{Msg}(id,\text{H},\text{CacheReq},a) \odot in,\ out)$

$\rightarrow\ \text{Msite}(mem,\ in,\ out)$

Figure 6.7 gives the imperative and directive rules used in the derivation for each Migratory rule. In the derivation, CachePending and T$[id]$ used in the integrated rules are mapped to Invalid and C$[id]$ of the imperative rules, respectively. Therefore, Migratory is a sound implementation of the CRF model.

Migratory Rule	Migratory Imperative & Directive Rules
P1	*Loadl-on-Clean*
P2	*Loadl-on-Dirty*
P3	ϵ
P4	*C-Send-CacheReq*
P5	*Storel-on-Clean*
P6	*Storel-on-Dirty*
P7	ϵ
P8	*C-Send-CacheReq*
P9	*Commit-on-Clean*
P10	*Commit-on-Dirty*
P11	ϵ
P12	*Commit-on-Invalid*
P13	*Reconcile-on-Clean*
P14	*Reconcile-on-Dirty*
P15	ϵ
P16	*Reconcile-on-Invalid*
VC1	*C-Send-Flush*
VC2	*C-Send-Purge*
VC3	*C-Send-CacheReq*
MC1	*C-Receive-Cache*
MC2	*C-Receive-Cache*
MC3	*C-Receive-FlushReq* + *C-Send-Purge*
MC4	*C-Receive-FlushReq* + *C-Send-Flush*
MC5	*C-Receive-FlushReq*
MC6	*C-Receive-FlushReq*
VM1	*M-Send-Cache*
VM2	*M-Send-FlushReq*
MM1	*M-Receive-CacheReq* + *M-Send-Cache*
MM2	*M-Send-FlushReq*
MM3	*M-Receive-CacheReq*
MM4	ϵ
MM5	*M-Receive-CacheReq*
MM6	*M-Receive-Purge*
MM7	*M-Receive-Purge*
MM8	*M-Receive-Flush*
MM9	*M-Receive-Flush*

Figure 6.7: Derivation of Migratory from Imperative & Directive Rules

6.5 Liveness Proof of the Migratory Protocol

We prove the liveness of Migratory by showing that each processor can always make progress. We assume FIFO message passing and proper buffer management as described in Section 6.3. The lemmas and theorems in this section are given in the context of the integrated rules of Migratory.

6.5.1 Some Invariants of Migratory

Lemma 44 is the same as Lemma 36, except that the C[id] state of the imperative rules is replaced by the C[id] and T[id] states of the integrated rules.

Lemma 44 Given a Migratory term s,

$$\begin{array}{l} \text{Cell}(a,v,\text{C}[id]) \in \text{Mem}(s) \;\vee\; \text{Cell}(a,v,\text{T}[id]) \in \text{Mem}(s) \quad \Leftrightarrow \\ \qquad \text{Cell}(a,v,\text{Clean}) \in \text{Cache}_{id}(s) \;\vee\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(s) \;\vee \\ \qquad \text{Msg}(\text{H},id,\text{Cache},a,v) \in \text{MoutCin}_{id}(s) \;\vee\; \text{Msg}(id,\text{H},\text{Purge},a) \in \text{MinCout}_{id}(s) \;\vee \\ \qquad \text{Msg}(id,\text{H},\text{Flush},a,\text{-}) \in \text{MinCout}_{id}(s) \end{array}$$

Lemma 45 includes invariants regarding message generation and processing. Invariants (1)-(2) ensure that when a Loadl or Storel instruction is performed on an uncached address, the cache will eventually contain a clean cell for the accessed address, or issue a CacheReq message and set the cache state to CachePending. The proof follows from the weak fairness of Rules P4 and P8.

Invariants (3)-(5) ensure that when a cache receives a Cache message, it will cache the data in the Clean state; when a cache receives a FlushReq message, if the address is cached in the Clean or Dirty state, the cache will send a Purge or Flush message to the memory. The proof follows from the weak fairness of Rules MC1, MC2, MC3 and MC4. Invariants (6)-(7) ensure that when the memory receives a Purge or Flush message, it will set the memory state to C[ϵ]. The proof follows from the weak fairness of Rules MM6, MM7, MM8 and MM9.

Invariants (8)-(9) ensure that each outgoing message will be delivered to its destination's incoming queue. This can be proved by induction on the number of preceding messages in the outgoing queue (note that the message passing rules are weakly fair).

Lemma 45 Given a Migratory sequence σ,

$$\begin{array}{ll}
(1) & \langle t,\text{Loadl}(a)\rangle \in \text{Pmb}_{id}(\sigma) \;\wedge\; a \notin \text{Cache}_{id}(\sigma) \quad \rightsquigarrow \\
& \qquad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee \\
& \qquad (\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \;\wedge\; \text{Msg}(id,\text{H},\text{CacheReq},a) \in \text{Cout}_{id}(\sigma)) \\
(2) & \langle t,\text{Storel}(a,\text{-})\rangle \in \text{Pmb}_{id}(\sigma) \;\wedge\; a \notin \text{Cache}_{id}(\sigma) \quad \rightsquigarrow \\
& \qquad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \;\vee \\
& \qquad (\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \;\wedge\; \text{Msg}(id,\text{H},\text{CacheReq},a) \in \text{Cout}_{id}(\sigma)) \\
(3) & \text{Msg}(\text{H},id,\text{Cache},a,\text{-})_{\uparrow} \in \text{Cin}_{id}(\sigma) \quad \rightsquigarrow \quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \\
(4) & \text{Msg}(\text{H},id,\text{FlushReq},a)_{\uparrow} \in \text{Cin}_{id}(\sigma) \;\wedge\; \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \quad \rightsquigarrow \\
& \qquad \text{Msg}(id,\text{H},\text{Purge},a) \in \text{Cout}_{id}(\sigma) \\
(5) & \text{Msg}(\text{H},id,\text{FlushReq},a)_{\uparrow} \in \text{Cin}_{id}(\sigma) \;\wedge\; \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \quad \rightsquigarrow \\
& \qquad \text{Msg}(id,\text{H},\text{Flush},a,\text{-}) \in \text{Cout}_{id}(\sigma) \\
(6) & \text{Msg}(id,\text{H},\text{Flush},a,\text{-})_{\uparrow} \in \text{Min}(\sigma) \quad \rightsquigarrow \quad \text{Cell}(a,\text{-},\text{C}[\epsilon]) \in \text{Mem}(\sigma) \\
(7) & \text{Msg}(id,\text{H},\text{Purge},a)_{\uparrow} \in \text{Min}(\sigma) \quad \rightsquigarrow \quad \text{Cell}(a,\text{-},\text{C}[\epsilon]) \in \text{Mem}(\sigma) \\
(8) & msg \in \text{Cout}_{id}(\sigma) \;\wedge\; \text{Dest}(msg) = \text{H} \quad \rightsquigarrow \quad msg \in \text{Min}(\sigma) \\
(9) & msg \in \text{Mout}(\sigma) \;\wedge\; \text{Dest}(msg) = id \quad \rightsquigarrow \quad msg \in \text{Cin}_{id}(\sigma)
\end{array}$$

Lemma 46 ensures that an incoming Cache, FlushReq, Purge or Flush message will eventually become the first message regarding the address in the incoming queue. Therefore, the message will be brought to the front end of the incoming queue sooner or later so that it can be processed. This lemma is a special case of Lemma 48.

Lemma 46 Given a Migratory sequence σ,

(1) $\quad msg \in \mathrm{Cin}_{id}(\sigma) \quad \rightsquigarrow \quad msg_{\uparrow} \in \mathrm{Cin}_{id}(\sigma)$

(2) $\quad msg \in \mathrm{Min}(\sigma) \ \wedge \ (\mathrm{Cmd}(msg) = \mathrm{Purge} \ \vee \ \mathrm{Cmd}(msg) = \mathrm{Flush}) \quad \rightsquigarrow \quad msg_{\uparrow} \in \mathrm{Min}(\sigma)$

Proof The proof is based on induction on the number of preceding messages in the incoming queue. Obviously the first message in a cache's incoming queue can always be processed because of the weak fairness of the mandatory cache engine rules. At the memory side, the first incoming message can always be processed except that a CacheReq message may need to be stalled. The critical observation here is that a CacheReq message followed by a Purge or Flush message cannot be stalled. This is because according to Lemma 44, the memory state shows the address is cached in the cache, which implies that the CacheReq message can be discarded according to Rule MM3 or MM5. □

Lemma 47 ensures that if a memory cell is in a transient state, it will eventually become C[ϵ]. This is crucial to ensure that stalled CacheReq messages will be processed.

Lemma 47 Given a Migratory sequence σ,

$\mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \in \mathrm{Mem}(\sigma) \quad \rightsquigarrow \quad \mathrm{Cell}(a,\text{-},\mathrm{C}[\epsilon]) \in \mathrm{Mem}(\sigma)$

Proof We first show some properties that are needed for the proof. All these properties can be verified by simply checking the Migratory rules.

- The memory sends a FlushReq message to cache site id whenever it changes the state of a memory cell to T[id].

  $\mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \notin \mathrm{Mem}(\sigma) \ \wedge \ \bigcirc\, \mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \in \mathrm{Mem}(\sigma)$
  $\quad \Rightarrow \quad \bigcirc\, \mathrm{Msg}(\mathrm{H},id,\mathrm{FlushReq},a)_{\downarrow} \in \mathrm{Mout}(\sigma)$

- The memory cannot change the memory state from T[id] to a different state unless it receives a Purge or Flush message from the cache site.

  $\mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \in \mathrm{Mem}(\sigma) \ \wedge \ \bigcirc\, \mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \notin \mathrm{Mem}(\sigma)$
  $\quad \Rightarrow \quad \mathrm{Msg}(id,\mathrm{H},\mathrm{Purge},a) \in \mathrm{Min}(\sigma) \ \vee \ \mathrm{Msg}(id,\mathrm{H},\mathrm{Flush},a,\text{-}) \in \mathrm{Min}(\sigma)$

- The T[id] state can only be changed to the C[ϵ] state.

  $\mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \in \mathrm{Mem}(\sigma) \ \wedge \ \bigcirc\, \mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \notin \mathrm{Mem}(\sigma)$
  $\quad \Rightarrow \quad \bigcirc\, \mathrm{Cell}(a,\text{-},\mathrm{C}[id]) \in \mathrm{Mem}(\sigma)$

- The memory cannot send any message to cache site id while the memory state is T[id]. This implies that if a message has no following message, it will remain as the last message.

  $\mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \in \mathrm{Mem}(\sigma) \ \wedge \ msg_{\downarrow} \in \mathrm{MoutCin}_{id}(\sigma)$
  $\quad \Rightarrow \quad \bigcirc\, msg_{\downarrow} \in \mathrm{MoutCin}_{id}(\sigma)$

We then prove the lemma under the assumption that the memory's outgoing queue contains a FlushReq message and the FlushReq message has no following message.

- According to Theorem-C and Lemma 45,

$$\begin{aligned}
&\mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \in \mathrm{Mem}(\sigma) \;\wedge\; \mathrm{Msg}(\mathrm{H},id,\mathrm{FlushReq},a)_{\downarrow} \in \mathrm{Mout}(\sigma)\\
\leadsto\quad &(\mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \in \mathrm{Mem}(\sigma) \;\wedge\; \mathrm{Msg}(\mathrm{H},id,\mathrm{FlushReq},a)_{\downarrow} \in \mathrm{Cin}_{id}(\sigma)) \;\vee\\
&\mathrm{Cell}(a,\text{-},\mathrm{C}[\epsilon]) \in \mathrm{Mem}(\sigma)
\end{aligned}$$

- According to Theorem-C and Lemma 46,

$$\begin{aligned}
&\mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \in \mathrm{Mem}(\sigma) \;\wedge\; \mathrm{Msg}(\mathrm{H},id,\mathrm{FlushReq},a)_{\downarrow} \in \mathrm{Cin}_{id}(\sigma)\\
\leadsto\quad &(\mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \in \mathrm{Mem}(\sigma) \;\wedge\; \mathrm{Msg}(\mathrm{H},id,\mathrm{FlushReq},a)_{\updownarrow} \in \mathrm{Cin}_{id}(\sigma)) \;\vee\\
&\mathrm{Cell}(a,\text{-},\mathrm{C}[id]) \in \mathrm{Mem}(\sigma)
\end{aligned}$$

- According to Lemmas 44, 45, and 46,

$$\begin{aligned}
&\mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \in \mathrm{Mem}(\sigma) \;\wedge\; \mathrm{Msg}(\mathrm{H},id,\mathrm{FlushReq},a)_{\updownarrow} \in \mathrm{Cin}_{id}(\sigma)\\
\leadsto\quad &\mathrm{Msg}(id,\mathrm{H},\mathrm{Purge},a) \in \mathrm{MinCout}_{id}(\sigma) \;\vee\; \mathrm{Msg}(id,\mathrm{H},\mathrm{Flush},a,\text{-}) \in \mathrm{MinCout}_{id}(\sigma)\\
\leadsto\quad &\mathrm{Msg}(id,\mathrm{H},\mathrm{Purge},a) \in \mathrm{Min}(\sigma) \;\vee\; \mathrm{Msg}(id,\mathrm{H},\mathrm{Flush},a,\text{-}) \in \mathrm{Min}(\sigma)\\
\leadsto\quad &\mathrm{Msg}(id,\mathrm{H},\mathrm{Purge},a)_{\uparrow} \in \mathrm{Min}(\sigma) \;\vee\; \mathrm{Msg}(id,\mathrm{H},\mathrm{Flush},a,\text{-})_{\uparrow} \in \mathrm{Min}(\sigma)\\
\leadsto\quad &\mathrm{Cell}(a,\text{-},\mathrm{C}[\epsilon]) \in \mathrm{Mem}(\sigma)
\end{aligned}$$

  Thus, $\mathrm{Cell}(a,\text{-},\mathrm{T}[id]) \in \mathrm{Mem}(\sigma) \;\wedge\; \mathrm{Msg}(\mathrm{H},id,\mathrm{FlushReq},a)_{\downarrow} \in \mathrm{Mout}(\sigma)$
  $\leadsto\quad \mathrm{Cell}(a,\text{-},\mathrm{C}[\epsilon]) \in \mathrm{Mem}(\sigma)$

This completes the proof according to Theorem-A. □

The proof above can be described in a more intuitive way. The memory sends a FlushReq message to cache site *id* each time it changes the memory state to T[*id*]. When the FlushReq message is received at the cache, if the cache state is Clean or Dirty, the cache sends a Purge or Flush message to the memory; otherwise the cache ignores the FlushReq message. In the latter case, if the memory state remains as T[*id*], there is a Purge or Flush message in transit from the cache to the memory according to Lemma 44 (note that the memory cannot send any message to the cache while the memory state is T[*id*], and FIFO message passing guarantees that any preceding message has been received before the Purge or Flush message is received). When the memory receives the Purge or Flush message, it sets the memory state to C[ϵ].

Lemma 48 is a general form of Lemma 46. It ensures that an incoming message will eventually become a message that has no preceding message. Note that Lemma 47 ensures that a stalled CacheReq message will be processed eventually, since Rule MM1 is strongly fair.

Lemma 48 Given a Migratory sequence σ,

(1) $\quad msg \in \mathrm{Cin}_{id}(\sigma) \quad\leadsto\quad msg_{\uparrow} \in \mathrm{Cin}_{id}(\sigma)$
(2) $\quad msg \in \mathrm{Min}(\sigma) \quad\leadsto\quad msg_{\uparrow} \in \mathrm{Min}(\sigma)$

Lemma 49 ensures that if an address is cached in the CachePending state, the cache state will become Clean eventually.

Lemma 49 Given a Migratory sequence σ,

$$\mathrm{Cell}(a,\text{-},\mathrm{CachePending}) \in \mathrm{Cache}_{id}(\sigma) \quad\leadsto\quad \mathrm{Cell}(a,\text{-},\mathrm{Clean}) \in \mathrm{Cache}_{id}(\sigma)$$

Proof We first show some properties that are needed for the proof. All these properties can be verified by simply checking the Migratory rules.

- A cache sends a CacheReq message to the memory whenever it changes the state of a cache cell to CachePending.

  $$\begin{aligned}&\text{Cell}(a,\text{-},\text{CachePending}) \notin \text{Cache}_{id}(\sigma) \ \wedge\ \bigcirc\, \text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma)\\ &\qquad\Rightarrow\quad \bigcirc\, \text{Msg}(id,\text{H},\text{CacheReq},a)_{\downarrow} \in \text{Cout}_{id}(\sigma)\end{aligned}$$

- The CachePending state can only be changed to the Clean state.

  $$\begin{aligned}&\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \ \wedge\ \bigcirc\, \text{Cell}(a,\text{-},\text{CachePending}) \notin \text{Cache}_{id}(\sigma)\\ &\qquad\Rightarrow\quad \bigcirc\, \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)\end{aligned}$$

- A cache cannot send any message to the memory while the cache state is CachePending.

  $$\begin{aligned}&\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \ \wedge\ msg_{\downarrow} \in \text{MinCout}_{id}(\sigma)\\ &\qquad\Rightarrow\quad \bigcirc\, msg_{\downarrow} \in \text{MinCout}_{id}(\sigma)\end{aligned}$$

We then prove the lemma under the assumption that the cache's outgoing queue contains a CacheReq message and the CacheReq message has no following message.

- According to Theorem-C and Lemma 45,

  $$\begin{aligned}&\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \ \wedge\ \text{Msg}(id,\text{H},\text{CacheReq},a)_{\downarrow} \in \text{Cout}_{id}(\sigma)\\ \leadsto\ \ &(\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \ \wedge\ \text{Msg}(id,\text{H},\text{CacheReq},a)_{\downarrow} \in \text{Min}(\sigma)) \ \vee\\ &\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)\end{aligned}$$

- According to Theorem-C and Lemma 48,

  $$\begin{aligned}&\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \ \wedge\ \text{Msg}(id,\text{H},\text{CacheReq},a)_{\downarrow} \in \text{Min}(\sigma)\\ \leadsto\ \ &(\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \ \wedge\ \text{Msg}(id,\text{H},\text{CacheReq},a)_{\updownarrow} \in \text{Min}(\sigma)) \ \vee\\ &\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)\end{aligned}$$

- According to Lemmas 44, 45, and 48,

  $$\begin{aligned}&\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \ \wedge\ \text{Msg}(id,\text{H},\text{CacheReq},a)_{\updownarrow} \in \text{Min}(\sigma)\\ \leadsto\ \ &\text{Msg}(\text{H},id,\text{Cache},a,\text{-}) \in \text{MoutCin}_{id}(\sigma)\\ \leadsto\ \ &\text{Msg}(\text{H},id,\text{Cache},a,\text{-}) \in \text{Cin}_{id}(\sigma)\\ \leadsto\ \ &\text{Msg}(\text{H},id,\text{Cache},a,\text{-})_{\uparrow} \in \text{Cin}_{id}(\sigma)\\ \leadsto\ \ &\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)\end{aligned}$$

  Thus, $\text{Cell}(a,\text{-},\text{CachePending}) \in \text{Cache}_{id}(\sigma) \ \wedge\ \text{Msg}(id,\text{H},\text{CacheReq},a)_{\downarrow} \in \text{Cout}_{id}(\sigma)$

  $\leadsto\quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma)$

This completes the proof according to Theorem-A. □

The proof above can be described in a more intuitive way. A cache generates a CacheReq message each time it changes the cache state of an address to CachePending. When the memory receives a CacheReq message from cache *id*, if the memory state is C[ϵ], the memory sends a Cache message to cache *id*; if the memory state is C[*id*] or T[*id*], the memory ignores the

CacheReq message. In the latter case, if the cache state remains as CachePending, there is a Cache message in transit from the memory to cache *id* according to Lemma 44 (note that the CacheReq message has no following message since the cache cannot issue any message while the cache state is CachePending, and FIFO message passing guarantees that any preceding message has been received before the CacheReq message is received). When the cache receives the Cache message, it caches the data and sets the cache state to Clean. It is worth pointing out that although the CacheReq message can be stalled at the memory, the stalled message will be processed eventually.

6.5.2 Liveness of Migratory

Lemma 50 ensures that whenever a processor intends to execute an instruction, the cache cell will be set to an appropriate state eventually. For a Loadl or Storel, the cache state will be set to Clean or Dirty; for a Commit or Reconcile, the cache state will be set to Clean, Dirty or Invalid.

Lemma 50 Given a Migratory sequence σ,

(1) $\text{Loadl}(a) \in \text{Pmb}_{id}(\sigma) \leadsto \text{Loadl}(a) \in \text{Pmb}_{id}(\sigma) \wedge$
$(\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma))$

(2) $\text{Storel}(a,\text{-}) \in \text{Pmb}_{id}(\sigma) \leadsto \text{Storel}(a,\text{-}) \in \text{Pmb}_{id}(\sigma) \wedge$
$(\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma))$

(3) $\text{Commit}(a) \in \text{Pmb}_{id}(\sigma) \leadsto \text{Commit}(a) \in \text{Pmb}_{id}(\sigma) \wedge$
$(\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \vee a \notin \text{Cache}_{id}(\sigma))$

(4) $\text{Reconcile}(a) \in \text{Pmb}_{id}(\sigma) \leadsto \text{Reconcile}(a) \in \text{Pmb}_{id}(\sigma) \wedge$
$(\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \vee a \notin \text{Cache}_{id}(\sigma))$

Proof We first show that whenever a processor intends to execute an instruction, the cache cell will be set to an appropriate state so that the instruction can be completed. This can be represented by the following proposition; the proof follows from Lemmas 45 and 49.

- $\text{Loadl}(a) \in \text{Pmb}_{id}(\sigma) \leadsto \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma)$
- $\text{Storel}(a,\text{-}) \in \text{Pmb}_{id}(\sigma) \leadsto \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma)$
- $\text{Commit}(a) \in \text{Pmb}_{id}(\sigma) \leadsto$
  $\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \vee a \notin \text{Cache}_{id}(\sigma)$
- $\text{Reconcile}(a) \in \text{Pmb}_{id}(\sigma) \leadsto$
  $\text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \vee \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \vee a \notin \text{Cache}_{id}(\sigma)$

We then show that an instruction can be completed only when the address is cached in an appropriate state. This can be represented by the following proposition, which can be verified by simply checking the Migratory rules that allow an instruction to be retired.

- $\langle t,\text{Loadl}(a)\rangle \in \text{Pmb}_{id}(\sigma) \ \wedge\ \circ\,\langle t,\text{Loadl}(a)\rangle \notin \text{Pmb}_{id}(\sigma)$
  $\Rightarrow\quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \ \vee\ \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma)$
- $\langle t,\text{Storel}(a,\text{-})\rangle \in \text{Pmb}_{id}(\sigma) \ \wedge\ \circ\,\langle t,\text{Storel}(a,\text{-})\rangle \notin \text{Pmb}_{id}(\sigma)$
  $\Rightarrow\quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \ \vee\ \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma)$
- $\langle t,\text{Commit}(a)\rangle \in \text{Pmb}_{id}(\sigma) \ \wedge\ \circ\,\langle t,\text{Commit}(a)\rangle \notin \text{Pmb}_{id}(\sigma)$
  $\Rightarrow\quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \ \vee\ \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \ \vee\ a \notin \text{Cache}_{id}(\sigma)$
- $\langle t,\text{Reconcile}(a)\rangle \in \text{Pmb}_{id}(\sigma) \ \wedge\ \circ\,\langle t,\text{Reconcile}(a)\rangle \notin \text{Pmb}_{id}(\sigma)$
  $\Rightarrow\quad \text{Cell}(a,\text{-},\text{Clean}) \in \text{Cache}_{id}(\sigma) \ \vee\ \text{Cell}(a,\text{-},\text{Dirty}) \in \text{Cache}_{id}(\sigma) \ \vee\ a \notin \text{Cache}_{id}(\sigma)$

This completes the proof according to Theorem-B. □

Lemma 50 ensures that when an instruction appears at the front of the processor-to-memory buffer, the cache cell will be brought into an appropriate state so that the instruction can be completed. The instruction will be completed eventually because of the strong fairness of Rules P1, P2, P5, P6, P9, P10, P12, P13, P14 and P16.

Theorem 51 (Liveness of Migratory) Given a Migratory sequence σ,

(1) $\langle t,\text{Loadl}(\text{-})\rangle \in \text{Pmb}_{id}(\sigma) \quad \rightsquigarrow \quad \langle t,\text{-}\rangle \in \text{Mpb}_{id}(\sigma)$
(2) $\langle t,\text{Storel}(\text{-},\text{-})\rangle \in \text{Pmb}_{id}(\sigma) \quad \rightsquigarrow \quad \langle t,\text{Ack}\rangle \in \text{Mpb}_{id}(\sigma)$
(3) $\langle t,\text{Commit}(\text{-})\rangle \in \text{Pmb}_{id}(\sigma) \quad \rightsquigarrow \quad \langle t,\text{Ack}\rangle \in \text{Mpb}_{id}(\sigma)$
(4) $\langle t,\text{Reconcile}(\text{-})\rangle \in \text{Pmb}_{id}(\sigma) \quad \rightsquigarrow \quad \langle t,\text{Ack}\rangle \in \text{Mpb}_{id}(\sigma)$

Chapter 7

Cachet: A Seamless Integration of Multiple Micro-protocols

The Cachet protocol is a seamless integration of the Base, WP and Migratory protocols that are presented in Chapters 4, 5 and 6. Although each protocol is complete in terms of functionality, we often refer to them as micro-protocols because they constitute parts of the full Cachet protocol. The Cachet protocol provides both intra-protocol and inter-protocol adaptivity that can be exploited via appropriate heuristic mechanisms to achieve optimal performance under changing program behaviors. Different micro-protocols can be used by different cache engines, and a cache can dynamically switch from one micro-protocol to another.

We first discuss the integration of micro-protocols, and the dynamic protocol switch through downgrade and upgrade operations. Section 7.2 describes the coherence states and protocol messages of Cachet. In Sections 7.3 and 7.4, we present the imperative and integrated rules of the Cachet protocol, respectively. Section 7.5 gives some composite rules that can be used to improve the performance without affecting the soundness and liveness of the system. The Cachet protocol assumes FIFO message passing, which requires that messages between the same source and destination be received in the order they are issued.

7.1 Integration of Micro-protocols

The CRF model allows a cache coherence protocol to use any cache or memory in the memory hierarchy as the rendezvous for processors that access shared memory locations, provided that it maintains the same observable behavior. The Base, WP and Migratory protocols are distinctive in the actions performed while committing dirty cells and reconciling clean cells. Figure 7.1 summarizes the different treatment of commit, reconcile and cache miss in the three micro-protocols.

Base: The most straightforward implementation simply uses the memory as the rendezvous. When a Commit instruction is executed for an address that is cached in the Dirty state, the data must be written back to the memory before the instruction can complete. A Reconcile

Micro-protocol	**Commit on Dirty**	**Reconcile on Clean**	**Cache Miss**
Base	update memory	purge local clean copy	retrieve data from memory
Writer-Push	purge all clean copies update memory		retrieve data from memory
Migratory			flush exclusive copy update memory retrieve data from memory

Figure 7.1: Different Treatment of Commit, Reconcile and Cache Miss

instruction for an address cached in the Clean state requires the data be purged from the cache before the instruction can complete. An attractive characteristic of Base is its simplicity; no state needs to be maintained at the memory side.

WP: Since load operations are usually more frequent than store operations, it is desirable to allow a Reconcile instruction to complete even when the address is cached in the Clean state. Thus, the following load access to the address causes no cache miss. Correspondingly, when a Commit instruction is performed on a dirty cell, it cannot complete before clean copies of the address are purged from all other caches. Therefore, it can be a lengthy process to commit an address that is cached in the Dirty state.

Migratory: When an address is exclusively accessed by one processor for a reasonable time period, it makes sense to give the cache the exclusive ownership so that all instructions on the address become local operations. This is reminiscent of the exclusive state in conventional invalidate-based protocols. The protocol ensures that an address can be cached in at most one cache at any time. Therefore, a Commit instruction can complete even when the address is cached in the Dirty state, and a Reconcile instruction can complete even when the address is cached in the Clean state. The exclusive ownership can migrate among different caches whenever necessary.

Different micro-protocols are optimized for different access patterns. The Base protocol is ideal when the location is randomly accessed by multiple processors and only necessary commit and reconcile operations are invoked. The WP protocol is appropriate when certain processors are likely to read an address many times before another processor writes the address. A reconcile operation performed on a clean copy causes no purge operation, regardless of whether the reconcile is necessary. Thus, subsequent load operations to the address can continually use the cached data without causing any cache miss. The Migratory protocol fits well when one processor is likely to read and write an address many times before another processor accesses the address.

7.1.1 Putting Things Together

It is easy to see that different addresses can employ different micro-protocols without any interference. The primary objective of Cachet is to integrate the micro-protocols in a seamless fashion in that different caches can use different micro-protocols on the same address simultaneously, and a cache can dynamically switch from one micro-protocol to another. For example, when something is known about the access pattern for a specific memory region, a cache can employ an appropriate micro-protocol for that region.

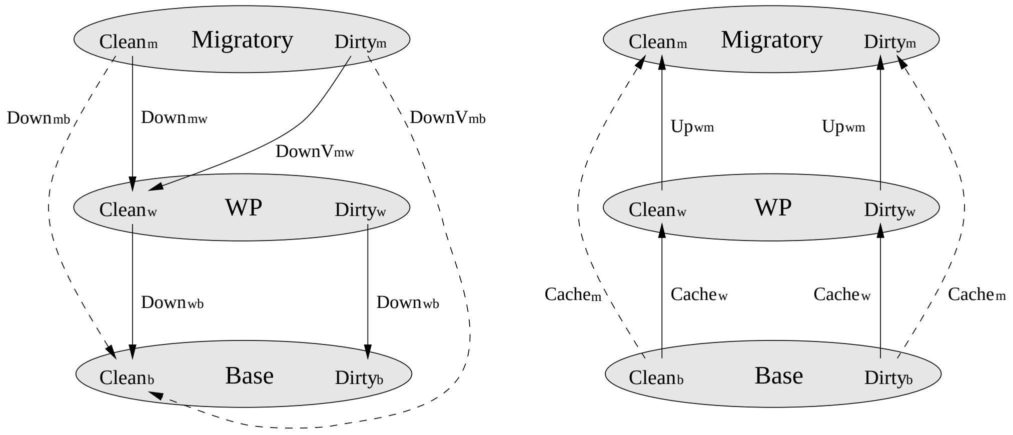
Different micro-protocols have different implications on the execution of memory instructions. The micro-protocols form an access privilege hierarchy. The Migratory protocol has the most privilege in that both commit and reconcile operations have no impact on the cache cell, while the Base protocol has the least privilege in that both commit and reconcile operations may require proper actions to be taken on the cache cell. The WP protocol has less privilege than Migratory but more privilege than Base. It allows reconcile operations to complete in the presence of a clean copy, but requires a dirty copy to be written back before commit operations on that address can complete.

With appropriate handling, the Base protocol can coexist with either WP or Migratory on the same address. The Base protocol requires that a dirty be written back to the memory on a commit, and a clean copy be purged on a reconcile so that the subsequent load operation must retrieve the data from the memory. This gives the memory an opportunity to take appropriate actions whenever necessary, regardless of how the address is cached in other caches at the time. In contrast, WP and Migratory cannot coexist with each other on the same address.

Since different micro-protocols have different treatment for Commit and Reconcile instructions, a cache must be able to tell which micro-protocol is in use for each cache cell. We can annotate a cache state with a subscript to represent the operational micro-protocol: $\mathrm{Clean_b}$ and $\mathrm{Dirty_b}$ are Base states, $\mathrm{Clean_w}$ and $\mathrm{Dirty_w}$ are WP states, and $\mathrm{Clean_m}$ and $\mathrm{Dirty_m}$ are Migratory states. The Cachet protocol draws no distinction between different micro-protocols for an uncached address, or an address cached in a transient state. We can also use subscripts to distinguish protocol messages whenever necessary. For example, the memory can supply a Base, WP or Migratory copy to a cache via a $\mathrm{Cache_b}$, $\mathrm{Cache_w}$ or $\mathrm{Cache_m}$ message. A cache can write a dirty copy back to the memory via a $\mathrm{Wb_b}$ or $\mathrm{Wb_w}$ message, depending on whether Base or WP is in use on the address.

7.1.2 Dynamic Micro-protocol Switch

The Cachet protocol provides inter-protocol adaptivity via downgrade and upgrade operations. A downgrade operation switches a cache cell to a less privileged micro-protocol, while an upgrade operation switches a cache cell to a more privileged micro-protocol. Figure 7.2 shows the cache state transitions caused by downgrade and upgrade operations (associated with each transition is the corresponding protocol message that is generated or received at the cache site).



(a) Downgrade Operations

(b) Upgrade Operations

Figure 7.2: Downgrade and Upgrade Operations

Downgrade Operations There are three types of downgrade operations, WP-to-Base (WB), Migratory-to-WP (MW) and Migratory-to-Base (MB). A cache can downgrade a cache cell from WP to Base, or from Migratory to WP or Base. The Migratory-to-Base downgrade is a composite operation equivalent to a Migratory-to-WP downgrade followed by a WP-to-Base downgrade.

When a cache downgrades a dirty Migratory cell, it always writes the data back to the memory. This ensures that the memory contains the most up-to-date data for any address that is not cached under Migratory. Consequently, Migratory cells can only be downgraded to clean Base or clean WP cells. A downgrade operation can happen voluntarily, or mandatorily upon a request from the memory. When a cache cell is downgraded, the cache sends a message to inform the memory of the downgrade operation.

Upgrade Operations The memory can send a message to upgrade a cache cell from WP to Migratory, or from Base to WP or Migratory. The Base-to-Migratory upgrade is a composite operation equivalent to a Base-to-WP upgrade followed by a WP-to-Migratory upgrade. Since the memory maintains no information about Base cells, it cannot draw distinction between the situation in which an address is not cached in a cache and the situation in which the address is cached in the cache under Base. Therefore, the Cache_w and Cache_m messages can behave as upgrade messages when the address is cached in a Base state at the destination cache.

Downgrade and upgrade operations may conflict with each other when they are performed on the same cache cell simultaneously. For example, a cache can downgrade a cache cell without consulting the memory, while the memory can send an upgrade message to upgrade the cache cell. In this case, the downgrade operation has higher priority in the sense that the cache cell will be eventually downgraded, and the upgrade message will be discarded when it is received.

Writeback Operations In Cachet, a writeback operation can also invoke a micro-protocol switch. The memory acknowledges each writeback operation with a WbAck_b message, except for the last writeback operation. The memory may acknowledge the last writeback operation with a WbAck_b, WbAck_w or WbAck_m message, which indicates that the cache can retain a Base, WP or Migratory copy after the writeback acknowledgment is received. For example, suppose a cache receives an acknowledgment regarding a writeback operation on a dirty WP copy. The cache can downgrade the cache cell, remain in WP, or upgrade the cache cell, depending on whether the acknowledgment is WbAck_b, WbAck_w or WbAck_m.

7.2 The System Configuration of the Cachet Protocol

7.2.1 Cache and Memory States

The Cachet protocol employs six stable cache states that represent the Clean and Dirty states of the three micro-protocols, and two transient cache states, WbPending and CachePending. In addition, the Invalid state corresponds to the Invalid states of the three micro-protocols.

- Clean_b: The Clean state of Base, which allows Commit instructions to complete but stalls Reconcile instructions.
- Dirty_b: The Dirty state of Base, which allows Reconcile instructions to complete but stalls Commit instructions.
- Clean_w: The Clean state of WP, which allows both Commit and Reconcile instructions to complete.
- Dirty_w: The Dirty state of WP, which allows Reconcile instructions to complete but stalls Commit instructions.
- Clean_m: The Clean state of Migratory, which allows both Commit and Reconcile instructions to complete.
- Dirty_m: The Dirty state of Migratory, which allows both Commit and Reconcile instructions to complete.
- WbPending: The transient state that indicates a dirty copy of the address is being written back to the memory.
- CachePending: The transient state that indicates a data copy is being requested for the address.
- Invalid: The address is not resident in the cache.

The Cachet protocol employs two stable memory states and three transient memory states.

- $\text{C}_\text{w}[dir]$: The address is cached under WP in the cache sites specified by the directory *dir*.
- $\text{C}_\text{m}[id]$: The address is cached under Migratory in cache site *id*.
- $\text{T}_\text{w}[dir,sm]$: The address is cached under WP in the cache sites specified by the directory *dir*; the suspended message buffer *sm* contains suspended writeback messages. The memory has multicast a DownReq_wb message to cache sites *dir*.

	Basic Messages	**Composite Messages**
Imperative Messages	$\text{Msg}(id,\text{H},\text{CacheReq},a)$ $\text{Msg}(id,\text{H},\text{Wb}_\text{b},a,v)$ $\text{Msg}(id,\text{H},\text{Down}_\text{wb},a)$ $\text{Msg}(id,\text{H},\text{Down}_\text{mw},a)$ $\text{Msg}(id,\text{H},\text{DownV}_\text{mw},a,v)$ $\text{Msg}(\text{H},id,\text{Cache}_\text{b},a,v)$ $\text{Msg}(\text{H},id,\text{Cache}_\text{w},a,v)$ $\text{Msg}(\text{H},id,\text{Up}_\text{wm},a)$ $\text{Msg}(\text{H},id,\text{WbAck}_\text{b},a)$ $\text{Msg}(\text{H},id,\text{WbAck}_\text{w},a)$	$\text{Msg}(id,\text{H},\text{Wb}_\text{w},a,v)$ $\text{Msg}(id,\text{H},\text{Down}_\text{mb},a)$ $\text{Msg}(id,\text{H},\text{DownV}_\text{mb},a,v)$ $\text{Msg}(\text{H},id,\text{Cache}_\text{m},a,v)$ $\text{Msg}(\text{H},id,\text{WbAck}_\text{m},a)$
Directive Messages	$\text{Msg}(\text{H},id,\text{DownReq}_\text{wb},a)$ $\text{Msg}(\text{H},id,\text{DownReq}_\text{mw},a)$	$\text{Msg}(\text{H},id,\text{DownReq}_\text{mb},a)$

Figure 7.3: Protocol Messages of Cachet

- $\text{T}_\text{m}[id,sm]$: The address is cached under Migratory in cache site id; the suspended message buffer sm contains suspended writeback messages. The memory has sent a DownReq_mb message (or a DownReq_mw followed by a DownReq_wb) to cache site id.
- $\text{T}'_\text{m}[id]$: The address is cached under Migratory in cache site id. The memory has sent a DownReq_mw message to cache site id.

The memory maintains no information about cache cells that use the Base micro-protocol. For example, the $\text{C}_\text{w}[\epsilon]$ state means that the address is not cached under WP or Migratory in any cache, but the address can still be cached under Base in some caches. Therefore, the memory should conservatively assume that a cache may contain a Base copy even though the memory state shows that the address is not resident in the cache.

It is also worth mentioning that $\text{C}_\text{w}[dir]$, $\text{C}_\text{m}[id]$ and $\text{T}'_\text{m}[id]$ are introduced purely for liveness reason, and are not used in imperative rules. In terms of soundness, $\text{C}_\text{w}[dir]$ is identical to $\text{T}_\text{w}[dir,\epsilon]$, while $\text{C}_\text{m}[id]$ and $\text{T}'_\text{m}[id]$ are identical to $\text{T}_\text{m}[id,\epsilon]$. This is obvious because downgrade requests are directive messages that are not used in the imperative rules.

7.2.2 Basic and Composite Messages

Protocol messages can be classified as imperative messages and directive messages. Imperative messages are used in the imperative rules that determine the soundness of the protocol, while directive messages are used to invoke proper imperative actions to ensure the liveness of the protocol. On a different dimension, protocol messages can be categorized as basic messages and composite messages. A basic message can be used to perform an operation that cannot be achieved by other messages, while a composite message is equivalent to piggybacked basic messages in that its behavior can be emulated by a sequence of basic messages. Figure 7.3 gives the protocol messages of Cachet.

There are ten basic imperative messages and two basic directive messages.

Composite Message	**Equivalent Sequence of Basic Messages**
Msg(id,H,Wb_{w},a,v)	Msg(id,H,Down_{wb},a) + Msg(id,H,Wb_{b},a,v)
Msg(id,H,Down_{mb},a)	Msg(id,H,Down_{mw},a) + Msg(id,H,Down_{wb},a)
Msg(id,H,DownV_{mb},a)	Msg(id,H,DownV_{mw},a) + Msg(id,H,Down_{wb},a)
Msg(H,id,Cache_{m},a,v)	Msg(H,id,Cache_{w},a,v) + Msg(H,id,Up_{wm},a)
Msg(H,id,WbAck_{m},a)	Msg(H,id,WbAck_{w},a) + Msg(H,id,Up_{wm},a)
Msg(H,id,$\text{DownReq}_{\text{mb}}$,$a$)	Msg(H,id,$\text{DownReq}_{\text{mw}}$,$a$) + Msg(H,$id$,$\text{DownReq}_{\text{wb}}$,$a$)

Figure 7.4: Composite Messages of Cachet

- CacheReq: The cache requests a data copy from the memory for an address.
- Wb_{b}: The cache writes the data of a dirty Base cell back to the memory.
- Down_{wb}: The cache informs the memory that a cache cell has been downgraded from WP to Base.
- Down_{mw}: The cache informs the memory that a clean cache cell has been downgraded from Migratory to WP.
- DownV_{mw}: The cache informs the memory that a dirty cache cell has been downgraded from Migratory to WP; the message also carries the modified data of the cell.
- Cache_{b}: The memory supplies a Base copy to the cache.
- Cache_{w}: The memory supplies a WP copy to the cache.
- Up_{wm}: The memory intends to upgrade a cache cell from WP to Migratory.
- WbAck_{b}: The memory acknowledges a writeback and allows the cache to retain a Base copy.
- WbAck_{w}: The memory acknowledges a writeback allows the cache to retain a WP copy.
- $\text{DownReq}_{\text{wb}}$: The memory requests a cache cell to downgrade from WP to Base.
- $\text{DownReq}_{\text{mw}}$: The memory requests a cache cell to downgrade from Migratory to WP.

There are five composite imperative messages and one composite directive message. Figure 7.4 shows the equivalent sequence of basic messages for each composite message.

- Wb_{w}: The cache writes the data of a dirty WP cell back to the memory. The message behaves as a Down_{wb} followed by a Wb_{b}.
- Down_{mb}: The cache informs the memory that a clean cache cell has been downgraded from Migratory to Base. The message behaves as a Down_{mw} followed by a Down_{wb}.
- DownV_{mb}: The cache informs the memory that a dirty cache cell has been downgraded from Migratory to Base. The message behaves as a DownV_{mw} followed by a Down_{wb}.
- Cache_{m}: The memory supplies a Migratory copy to the cache. The message behaves as a Cache_{w} followed by an Up_{wm}.
- WbAck_{m}: The memory acknowledges a writeback and allows the cache to retain a Migratory copy. The message behaves as a WbAck_{w} followed by an Up_{wm}.

- $\text{DownReq}_{\text{mb}}$: The memory requests a cache cell to downgrade from Migratory to Base. The message behaves as $\text{DownReq}_{\text{mw}}$ followed by a $\text{DownReq}_{\text{wb}}$.

In the remainder of this chapter, we first design Cachet with only the basic messages. We then define the complete Cachet protocol by incorporating composite operations that use composite messages. The incorporation of composite operations can improve the performance of the system but cannot compromise the soundness and liveness of the protocol. The notion of composite messages and composite operations can remarkably simplify the design and verification of sophisticated cache coherence protocols such as Cachet, since composite composite rules do not have to be considered throughout the verification. It suffices to verify the correctness in the presence of basic messages only.

7.3 The Imperative Rules of the Cachet Protocol

The imperative rules of Cachet contain three sets of rules, the processor rules, the cache engine rules and the memory engine rules.

7.3.1 Imperative Processor Rules

Figure 7.5 gives the imperative processor rules of Cachet. It includes the imperative processor rules from the Base, WP and Migratory protocols. When an instruction is retired, it is removed from the processor-to-memory buffer while an appropriate reply is provided via the memory-to-processor buffer. The reply can be the requested data for a Loadl instruction, or the corresponding acknowledgment for a Storel, Commit and Reconcile instruction.

- A Loadl instruction can be completed if the address is cached in the Clean or Dirty state of any protocol.
- A Storel instruction can be completed if the address is cached in the Clean or Dirty state of any protocol. The cache state is set to Dirty while the cache value is updated.
- A Commit instruction can be completed if the addressed is uncached, or cached in the Clean state of any protocol or the Dirty state of the Migratory protocol.
- A Reconcile instruction can be completed if the address is uncached, or cached in the Dirty state of any protocol or the Clean state of the WP or Migratory protocol.

7.3.2 Imperative Cache Engine Rules

Figure 7.6 gives the imperative C-engine rules of Cachet. When a cache engine processes an incoming message, it immediately removes the message from the incoming queue.

- A cache can purge a clean Base cell (Rule IC1).
- A cache can write the dirty data of a Base cell back to the memory via a Wb_{b} message and set the cache state to WbPending, indicating that a writeback operation is being

Imperative Processor Rules				
Instruction	**Cstate**	**Action**	**Next Cstate**	
Loadl(a)	Cell(a,v,Clean$_{\text{b}}$)	*retire*	Cell(a,v,Clean$_{\text{b}}$)	IP1
	Cell(a,v,Dirty$_{\text{b}}$)	*retire*	Cell(a,v,Dirty$_{\text{b}}$)	IP2
	Cell(a,v,Clean$_{\text{w}}$)	*retire*	Cell(a,v,Clean$_{\text{w}}$)	IP3
	Cell(a,v,Dirty$_{\text{w}}$)	*retire*	Cell(a,v,Dirty$_{\text{w}}$)	IP4
	Cell(a,v,Clean$_{\text{m}}$)	*retire*	Cell(a,v,Clean$_{\text{m}}$)	IP5
	Cell(a,v,Dirty$_{\text{m}}$)	*retire*	Cell(a,v,Dirty$_{\text{m}}$)	IP6
Storel(a,v)	Cell(a,-,Clean$_{\text{b}}$)	*retire*	Cell(a,v,Dirty$_{\text{b}}$)	IP7
	Cell(a,-,Dirty$_{\text{b}}$)	*retire*	Cell(a,v,Dirty$_{\text{b}}$)	IP8
	Cell(a,-,Clean$_{\text{w}}$)	*retire*	Cell(a,v,Dirty$_{\text{w}}$)	IP9
	Cell(a,-,Dirty$_{\text{w}}$)	*retire*	Cell(a,v,Dirty$_{\text{w}}$)	IP10
	Cell(a,-,Clean$_{\text{m}}$)	*retire*	Cell(a,v,Dirty$_{\text{m}}$)	IP11
	Cell(a,-,Dirty$_{\text{m}}$)	*retire*	Cell(a,v,Dirty$_{\text{m}}$)	IP12
Commit(a)	Cell(a,v,Clean$_{\text{b}}$)	*retire*	Cell(a,v,Clean$_{\text{b}}$)	IP13
	Cell(a,v,Clean$_{\text{w}}$)	*retire*	Cell(a,v,Clean$_{\text{w}}$)	IP14
	Cell(a,v,Clean$_{\text{m}}$)	*retire*	Cell(a,v,Clean$_{\text{m}}$)	IP15
	Cell(a,v,Dirty$_{\text{m}}$)	*retire*	Cell(a,v,Dirty$_{\text{m}}$)	IP16
	$a \notin$ *cache*	*retire*	$a \notin$ *cache*	IP17
Reconcile(a)	Cell(a,v,Dirty$_{\text{b}}$)	*retire*	Cell(a,v,Dirty$_{\text{b}}$)	IP18
	Cell(a,v,Clean$_{\text{w}}$)	*retire*	Cell(a,v,Clean$_{\text{w}}$)	IP19
	Cell(a,v,Dirty$_{\text{w}}$)	*retire*	Cell(a,v,Dirty$_{\text{w}}$)	IP20
	Cell(a,v,Clean$_{\text{m}}$)	*retire*	Cell(a,v,Clean$_{\text{m}}$)	IP21
	Cell(a,v,Dirty$_{\text{m}}$)	*retire*	Cell(a,v,Dirty$_{\text{m}}$)	IP22
	$a \notin$ *cache*	*retire*	$a \notin$ *cache*	IP23

Figure 7.5: Imperative Processor Rules of Cachet

performed (Rule IC2). The cache state will be set to Clean$_{\text{b}}$ or Clean$_{\text{w}}$ when the cache receives a writeback acknowledgment later (Rules IC21 and IC22).

- A cache can downgrade a cell from WP to Base, and send a Down$_{\text{wb}}$ message to the memory (Rules IC3 and IC4).
- A cache can downgrade a cell from Migratory to WP, and send a Down$_{\text{mw}}$ or DownV$_{\text{mw}}$ message to the memory (Rules IC5 and IC6). The most up-to-date data is always sent back to the memory when a dirty Migratory cell is downgraded.
- A cache can send a CacheReq message to the memory to request the data for an uncached address; the cache state is set to CachePending to indicate that a cache copy is being requested (Rule IC7).
- If a cache receives a Cache$_{\text{b}}$ message, it caches the data in the Clean state of Base (Rule IC8). Note that the memory cannot supply a Base copy without a request from the cache.
- If a cache receives a Cache$_{\text{w}}$ message for a clean Base cell, it updates the cache cell with the new data and upgrades the cache cell to WP (Rule IC9). This can happen because the memory maintains no information about Base cells. It is trivial to show that Rule IC9 can be derived from Rules IC1 and IC13.
- If a cache receives a Cache$_{\text{w}}$ message for a dirty Base cell, it upgrades the cache cell to WP (Rule IC10).
- If a cache receives a Cache$_{\text{w}}$ message for an address cached in the WbPending state, it discards the message (Rule IC11). This can happen when the cache writes the modified

Imperative C-engine Rules				
Msg from H	**Cstate**	**Action**	**Next Cstate**	
	Cell(a,-,Clean$_\text{b}$)		$a \notin cache$	IC1
	Cell(a,v,Dirty$_\text{b}$)	$\langle$Wb$_\text{b}$,a,$v\rangle \rightarrow$ H	Cell(a,v,WbPending)	IC2
	Cell(a,v,Clean$_\text{w}$)	$\langle$Down$_\text{wb}$,$a\rangle \rightarrow$ H	Cell(a,v,Clean$_\text{b}$)	IC3
	Cell(a,v,Dirty$_\text{w}$)	$\langle$Down$_\text{wb}$,$a\rangle \rightarrow$ H	Cell(a,v,Dirty$_\text{b}$)	IC4
	Cell(a,v,Clean$_\text{m}$)	$\langle$Down$_\text{mw}$,$a\rangle \rightarrow$ H	Cell(a,v,Clean$_\text{w}$)	IC5
	Cell(a,v,Dirty$_\text{m}$)	$\langle$DownV$_\text{mw}$,a,$v\rangle \rightarrow$ H	Cell(a,v,Clean$_\text{w}$)	IC6
	$a \notin cache$	$\langle$CacheReq,$a\rangle \rightarrow$ H	Cell(a,-,CachePending)	IC7
$\langle$Cache$_\text{b}$,a,$v\rangle$	Cell(a,-,CachePending)		Cell(a,v,Clean$_\text{b}$)	IC8
$\langle$Cache$_\text{w}$,a,$v\rangle$	Cell(a,-,Clean$_\text{b}$)		Cell(a,v,Clean$_\text{w}$)	IC9
	Cell(a,v_1,Dirty$_\text{b}$)		Cell(a,v_1,Dirty$_\text{w}$)	IC10
	Cell(a,v_1,WbPending)		Cell(a,v_1,WbPending)	IC11
	Cell(a,-,CachePending)		Cell(a,v,Clean$_\text{w}$)	IC12
	$a \notin cache$		Cell(a,v,Clean$_\text{w}$)	IC13
$\langle$Up$_\text{wm}$,$a\rangle$	Cell(a,v,Clean$_\text{b}$)		Cell(a,v,Clean$_\text{b}$)	IC14
	Cell(a,v,Dirty$_\text{b}$)		Cell(a,v,Dirty$_\text{b}$)	IC15
	Cell(a,v,Clean$_\text{w}$)		Cell(a,v,Clean$_\text{m}$)	IC16
	Cell(a,v,Dirty$_\text{w}$)		Cell(a,v,Dirty$_\text{m}$)	IC17
	Cell(a,v,WbPending)		Cell(a,v,WbPending)	IC18
	Cell(a,-,CachePending)		Cell(a,-,CachePending)	IC19
	$a \notin cache$		$a \notin cache$	IC20
$\langle$WbAck$_\text{b}$,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean$_\text{b}$)	IC21
$\langle$WbAck$_\text{w}$,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean$_\text{w}$)	IC22

Figure 7.6: Imperative Cache Engine Rules of Cachet

data of the Base cell back to the memory before it receives the Cache$_\text{w}$ message.

- If a cache receives a Cache$_\text{w}$ message for an uncached address, or an address cached in the CachePending state , it caches the data in the Clean state of WP (Rules IC12 and IC13).
- If a cache receives an Up$_\text{wm}$ message for a Base cell, it discards the message (Rules IC14 and IC15). This can happen because the cache can voluntarily downgrade a WP cell while the memory intends to upgrade the cache cell.
- If a cache receives an Up$_\text{wm}$ message for a WP cell, it upgrades the cache cell to Migratory (Rules 16 and 17).
- If a cache receives an Up$_\text{wm}$ message for an uncached address, or an address cached in the WbPending or CachePending state, it discards the message (Rules IC18, IC19 and IC20). This can happen if the cache has downgraded the cell from WP to Base before it receives the upgrade message, and the Base cell has been purged from the cache or written back to the memory.

Figure 7.7 shows the cache state transitions of Cachet due to imperative operations (composite operations are not shown).

7.3.3 Imperative Memory Engine Rules

Figure 7.8 gives the set of imperative M-engine rules of Cachet. When the memory engine receives a message, it removes the message from the incoming queue.

- If the memory state shows that an address is not resident in a cache, the memory can send a Cache$_\text{w}$ message to supply a WP copy to the cache (Rule IM1). The cache may contain

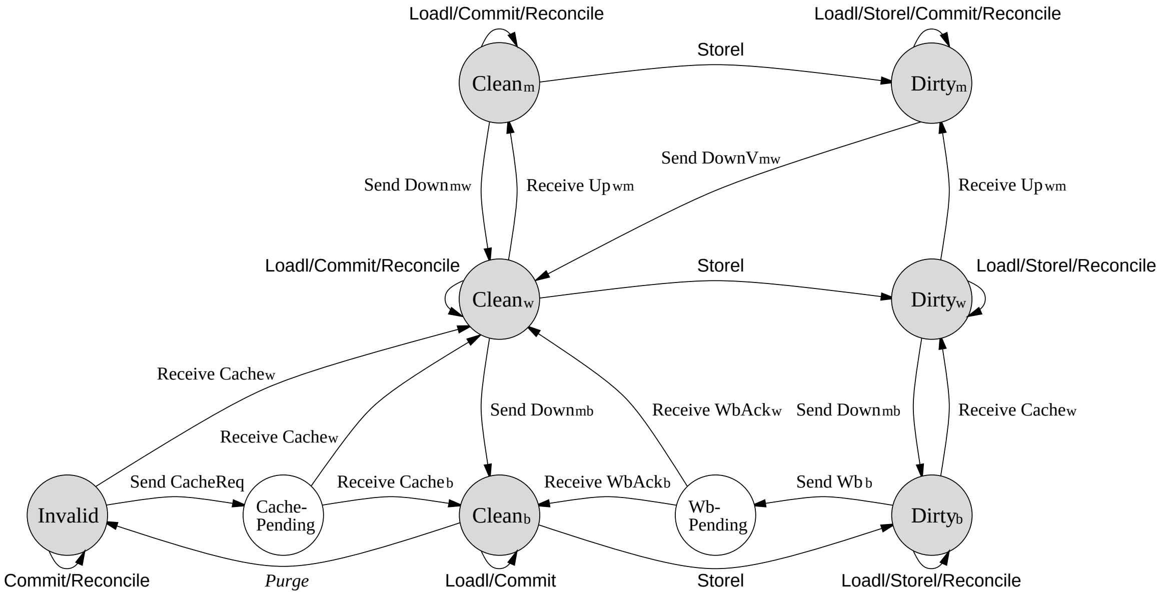

LoadI/Commit/Reconcile
LoadI/Storel/Commit/Reconcile
Storel
Clean$_{\mathrm{m}}$
Dirty$_{\mathrm{m}}$
Send DownV$_{\mathrm{mw}}$
Send Down$_{\mathrm{mw}}$
Receive Up$_{\mathrm{wm}}$
Receive Up$_{\mathrm{wm}}$
LoadI/Commit/Reconcile
Storel
LoadI/Storel/Reconcile
Clean$_{\mathrm{w}}$
Dirty$_{\mathrm{w}}$
Receive Cache$_{\mathrm{w}}$
Receive Cache$_{\mathrm{w}}$
Send Down$_{\mathrm{mb}}$
Receive WbAck$_{\mathrm{w}}$
Send Down$_{\mathrm{mb}}$
Receive Cache$_{\mathrm{w}}$
Send CacheReq
Receive Cache$_{\mathrm{b}}$
Receive WbAck$_{\mathrm{b}}$
Send Wb$_{\mathrm{b}}$
Invalid
Cache-Pending
Clean$_{\mathrm{b}}$
Wb-Pending
Dirty$_{\mathrm{b}}$
Commit/Reconcile
Purge
LoadI/Commit
Storel
LoadI/Storel/Reconcile

Figure 7.7: Cache State Transitions of Cachet

a Base cell, since the memory maintains no information for addresses cached under Base. In this case, the Cache$_{\mathrm{w}}$ message invokes an upgrade operation on the cache cell.

- If the memory state shows that an address is cached exclusively in a cache under WP, the memory can send an Up$_{\mathrm{wm}}$ message to upgrade the cache cell to Migratory (Rule IM2).
- When the memory receives a CacheReq message, if the memory state shows that the address is uncached in the cache and that the address is uncached in any other cache under Migratory, the memory can send a Cache$_{\mathrm{b}}$ message to supply a Base copy to the cache (Rule IM3). An alternative treatment is to let the memory send a Cache$_{\mathrm{w}}$ message to supply a WP copy (Rule IM1) and then discard the cache request (Rule IM4).
- When the memory receives a CacheReq message, if the memory state shows that the address is cached in the cache, the memory discards the cache request (Rules IM4 and IM5). This can happen because the memory can voluntarily supply a WP copy to a cache; the WP cell can be further upgraded to a Migratory cell.
- When the memory receives a Wb$_{\mathrm{b}}$ message, if the memory state shows that the address is uncached in the cache, the memory suspends the writeback message for later processing (Rules IM6 and IM8).
- When the memory receives a Wb$_{\mathrm{b}}$ message, if the memory state shows that the cache contains a WP copy for the address, the memory suspends the writeback message and removes the cache identifier from the corresponding directory (Rule IM7). This can happen if the memory sends a Cache$_{\mathrm{w}}$ message to the cache before it receives the Wb$_{\mathrm{b}}$ message. The Cache$_{\mathrm{w}}$ message will be discarded at the cache (Rule IC11).

Imperative M-engine Rules				
Msg from id	**Mstate**	**Action**	**Next Mstate**	
	$\text{Cell}(a,v,\text{T}_\text{w}[\mathit{dir},\epsilon])\ (\mathit{id} \notin \mathit{dir})$	$\langle \text{Cache}_\text{w},a,v\rangle \rightarrow \mathit{id}$	$\text{Cell}(a,v,\text{T}_\text{w}[\mathit{id} \mid \mathit{dir},\epsilon])$	IM1
	$\text{Cell}(a,v,\text{T}_\text{w}[\mathit{id},\epsilon])$	$\langle \text{Up}_\text{wm},a\rangle \rightarrow \mathit{id}$	$\text{Cell}(a,v,\text{T}_\text{m}[\mathit{id},\epsilon])$	IM2
$\langle \text{CacheReq},a\rangle$	$\text{Cell}(a,v,\text{T}_\text{w}[\mathit{dir},\mathit{sm}])\ (\mathit{id} \notin \mathit{dir})$	$\langle \text{Cache}_\text{b},a,v\rangle \rightarrow \mathit{id}$	$\text{Cell}(a,v,\text{T}_\text{w}[\mathit{dir},\mathit{sm}])$	IM3
	$\text{Cell}(a,v,\text{T}_\text{w}[\mathit{dir},\mathit{sm}])\ (\mathit{id} \in \mathit{dir})$		$\text{Cell}(a,v,\text{T}_\text{w}[\mathit{dir},\mathit{sm}])$	IM4
	$\text{Cell}(a,v,\text{T}_\text{m}[\mathit{id},\mathit{sm}])$		$\text{Cell}(a,v,\text{T}_\text{m}[\mathit{id},\mathit{sm}])$	IM5
$\langle \text{Wb}_\text{b},a,v\rangle$	$\text{Cell}(a,v_1,\text{T}_\text{w}[\mathit{dir},\mathit{sm}])\ (\mathit{id} \notin \mathit{dir})$		$\text{Cell}(a,v_1,\text{T}_\text{w}[\mathit{dir},(\mathit{id},v) \mid \mathit{sm}])$	IM6
	$\text{Cell}(a,v_1,\text{T}_\text{w}[\mathit{id} \mid \mathit{dir},\mathit{sm}])$		$\text{Cell}(a,v_1,\text{T}_\text{w}[\mathit{dir},(\mathit{id},v) \mid \mathit{sm}])$	IM7
	$\text{Cell}(a,v_1,\text{T}_\text{m}[\mathit{id}_1,\mathit{sm}])\ (\mathit{id} \neq \mathit{id}_1)$		$\text{Cell}(a,v_1,\text{T}_\text{m}[\mathit{id}_1,(\mathit{id},v) \mid \mathit{sm}])$	IM8
	$\text{Cell}(a,v_1,\text{T}_\text{m}[\mathit{id},\mathit{sm}])$		$\text{Cell}(a,v_1,\text{T}_\text{w}[\epsilon,(\mathit{id},v) \mid \mathit{sm}])$	IM9
$\langle \text{Down}_\text{wb},a\rangle$	$\text{Cell}(a,v,\text{T}_\text{w}[\mathit{id} \mid \mathit{dir},\mathit{sm}])$		$\text{Cell}(a,v,\text{T}_\text{w}[\mathit{dir},\mathit{sm}])$	IM10
	$\text{Cell}(a,v,\text{T}_\text{m}[\mathit{id},\mathit{sm}])$		$\text{Cell}(a,v,\text{T}_\text{w}[\epsilon,\mathit{sm}])$	IM11
$\langle \text{Down}_\text{mw},a\rangle$	$\text{Cell}(a,v,\text{T}_\text{m}[\mathit{id},\mathit{sm}])$		$\text{Cell}(a,v,\text{T}_\text{w}[\mathit{id},\mathit{sm}])$	IM12
$\langle \text{DownV}_\text{mw},a,v\rangle$	$\text{Cell}(a,\text{-},\text{T}_\text{m}[\mathit{id},\mathit{sm}])$		$\text{Cell}(a,v,\text{T}_\text{w}[\mathit{id},\mathit{sm}])$	IM13
	$\text{Cell}(a,\text{-},\text{T}_\text{w}[\epsilon,(\mathit{id},v) \mid \mathit{sm}])$	$\langle \text{WbAck}_\text{b},a\rangle \rightarrow \mathit{id}$	$\text{Cell}(a,v,\text{T}_\text{w}[\epsilon,\mathit{sm}])$	IM14
	$\text{Cell}(a,\text{-},\text{T}_\text{w}[\epsilon,(\mathit{id},v)])$	$\langle \text{WbAck}_\text{w},a\rangle \rightarrow \mathit{id}$	$\text{Cell}(a,v,\text{T}_\text{w}[\mathit{id},\epsilon])$	IM15

Figure 7.8: Imperative Memory Engine Rules of Cachet

- When the memory receives a Wb_b message, if the memory state shows that the cache contains a Migratory copy for the address, the memory suspends the writeback message and sets the memory state to indicate that the address is no longer cached in any cache (Rule IM9). This can happen if the memory sends a Cache_w message followed by an Up_wm message to the cache before it receives the Wb_b message. The Cache_w and Up_wm messages will be discarded at the cache (Rules IC11 and IC18).
- When the memory receives a Down_wb message, if the memory state shows that the cache contains a WP cell for the address, the memory removes the cache identifier from the directory (Rule IM10).
- When the memory receives a Down_wb message, if the memory state shows that the cache contains a Migratory cell for the address, the memory sets the memory state to indicate that the address is no longer cached in any cache (Rule IM11). This can happen when the memory sends an Up_wm message to the cache before it receives the Down_wb message. The Up_wm message will be discarded at the cache (Rules IC14, IC15, IC18, IC19 and IC20).
- When the memory receives a Down_mw or DownV_mw message, it sets the memory state accordingly to indicate that the cache cell has been downgraded to WP (Rules IM12 and IM13).
- When the memory state shows that an address is not resident in any cache, the memory can resume suspended writeback messages. For each writeback message, the memory updates the memory cell and acknowledges the cache via a WbAck_b message (Rule IM14). The last resumed message can be acknowledged with a WbAck_w message so that the cache can retain a WP copy (Rule IM15).

There are some subtle issues that must be handled properly to ensure correctness. As in Base, the memory cannot send a Cache_b message to a cache without a CacheReq message from the cache site. As in WP, the memory cannot be updated if the directory shows that the address is cached under WP in some caches. Since the memory maintains no information about Base

Composite Imperative C-engine Rules

Msg from H	**Cstate**	**Action**	**Next Cstate**	
	Cell(a,v,Dirty$_\mathrm{w}$)	$\langle$Wb$_\mathrm{w}$,a,$v\rangle \rightarrow$ H	Cell(a,v,WbPending)	CIC1
	Cell(a,v,Clean$_\mathrm{m}$)	$\langle$Down$_\mathrm{mb}$,$a\rangle \rightarrow$ H	Cell(a,v,Clean$_\mathrm{b}$)	CIC2
	Cell(a,v,Dirty$_\mathrm{m}$)	$\langle$DownV$_\mathrm{mb}$,a,$v\rangle \rightarrow$ H	Cell(a,v,Clean$_\mathrm{b}$)	CIC3
$\langle$Cache$_\mathrm{m}$,a,$v\rangle$	Cell(a,-,Clean$_\mathrm{b}$)		Cell(a,v,Clean$_\mathrm{m}$)	CIC4
	Cell(a,v_1,Dirty$_\mathrm{b}$)		Cell(a,v_1,Dirty$_\mathrm{m}$)	CIC5
	Cell(a,v_1,WbPending)		Cell(a,v_1,WbPending)	CIC6
	Cell(a,-,CachePending)		Cell(a,v,Clean$_\mathrm{m}$)	CIC7
	$a \notin cache$		Cell(a,v,Clean$_\mathrm{m}$)	CIC8
$\langle$WbAck$_\mathrm{m}$,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean$_\mathrm{m}$)	CIC9

Composite Imperative M-engine Rules

Msg from *id*	**Mstate**	**Action**	**Next Mstate**	
	Cell(a,v,T$_\mathrm{w}[\epsilon,\epsilon]$)	$\langle$Cache$_\mathrm{m}$,a,$v\rangle \rightarrow id$	Cell(a,v,T$_\mathrm{m}[id,\epsilon]$)	CIM1
$\langle$Wb$_\mathrm{w}$,a,$v\rangle$	Cell(a,v_1,T$_\mathrm{w}[id \mid dir,sm]$)		Cell(a,v_1,T$_\mathrm{w}[dir,(id,v) \mid sm]$)	CIM2
	Cell(a,v_1,T$_\mathrm{m}[id,sm]$)		Cell(a,v_1,T$_\mathrm{w}[\epsilon,(id,v) \mid sm]$)	CIM3
$\langle$Down$_\mathrm{mb}$,$a\rangle$	Cell(a,v,T$_\mathrm{m}[id,sm]$)		Cell(a,v,T$_\mathrm{w}[\epsilon,sm]$)	CIM4
$\langle$DownV$_\mathrm{mb}$,a,$v\rangle$	Cell(a,-,T$_\mathrm{m}[id,sm]$)		Cell(a,v,T$_\mathrm{w}[\epsilon,sm]$)	CIM5
	Cell(a,-,T$_\mathrm{w}[\epsilon,(id,v)]$)	$\langle$WbAck$_\mathrm{m}$,$a\rangle \rightarrow id$	Cell(a,v,T$_\mathrm{m}[id,\epsilon]$)	CIM6

Figure 7.9: Composite Imperative Rules of Cachet

cells, it may receive a writeback message from a cache although the memory state shows that the address is not resident in the cache. A writeback operation cannot be acknowledged before all WP and Migratory cells of the address have been purged or downgraded to Base cells.

7.3.4 Composite Imperative Rules

Figure 7.9 defines some composite imperative rules of Cachet. A composite imperative rule can be simulated by basic imperative rules, and therefore cannot compromise the soundness of the protocol.

Composite C-engine Rules

- A cache can write the dirty data of a WP cell back to the memory via a Wb$_\mathrm{w}$ message and set the cache state to WbPending (Rule CIC1). The Wb$_\mathrm{w}$ message behaves as a Down$_\mathrm{wb}$ followed by a Wb$_\mathrm{b}$.
- A cache can downgrade a cache cell from Migratory to Base, and inform the memory via a Down$_\mathrm{mb}$ or DownV$_\mathrm{mb}$ message (Rules CIC2 and CIC3). The Down$_\mathrm{mb}$ message behaves as a Down$_\mathrm{mw}$ followed by a Down$_\mathrm{wb}$, and the DownV$_\mathrm{mb}$ message behaves as a DownV$_\mathrm{mw}$ followed by a Down$_\mathrm{wb}$.
- If a cache receives a Cache$_\mathrm{m}$ message for a clean Base cell, it purges the cache cell and caches the data in the Clean state of Migratory (Rules CIC4).
- If a cache receives a Cache$_\mathrm{m}$ message for a dirty Base cell, it upgrades the cache cell to Migratory (Rule CIC5).
- If a cache receives a Cache$_\mathrm{m}$ message for an address cached in the WbPending state, it discards the message (Rule CIC6).

Composite Imperative Rule	**Simulating Imperative Rules**
CIC1	IC4 + IC2
CIC2	IC5 + IC3
CIC3	IC6 + IC3
CIC4	IC9 + IC16
CIC5	IC10 + IC17
CIC6	IC11 + IC18
CIC7	IC12 + IC16
CIC8	IC13 + IC16
CIC9	IC22 + IC16
CIM1	IM1 + IM2
CIM2	IM9 + IM6
CIM3	IM11 + IM6
CIM4	IM12 + IM10
CIM5	IM13 + IM10
CIM6	IM15 + IM2

Figure 7.10: Simulation of Composite Imperative Rules of Cachet

- If a cache receives a Cache$_{\rm m}$ message for an uncached address or an address cached in the CachePending state, it caches the data in the Clean state of Migratory (Rules CIC7 and CIC8).
- If a cache receives a WbAck$_{\rm m}$ message for an address cached in the WbPending state, it sets the cache state to Clean of Migratory (Rule CIC9).

Composite M-engine Rules

- If the memory state shows that an address is not cached in any cache, the memory can send a Cache$_{\rm m}$ message to supply a Migratory copy to a cache (Rule CIM1). The Cache$_{\rm m}$ message behaves as a Cache$_{\rm w}$ followed by an Up$_{\rm wm}$.
- When the memory receives a Wb$_{\rm w}$ message, if the memory state shows the cache contains a WP copy for the address, the memory removes the cache identifier from the directory and suspends the writeback message (Rule CIM2).
- When the memory receives a Wb$_{\rm w}$ message, if the memory state shows that the cache contains a Migratory copy for the address, the memory suspends the writeback message and sets the memory state to indicate that the address is no longer cached in any cache (Rule CIM3).
- When the memory receives a Down$_{\rm mb}$ or DownV$_{\rm mb}$ message, it updates the memory state to indicate that the address is no longer cached in any cache (Rules CIM4 and CIM5).
- When the memory state shows that an address is not resident in any cache and there is only one suspended writeback message, the memory can acknowledge the writeback message with a WbAck$_{\rm m}$ message to allow the cache to retain a Migratory copy (Rule CIM6).

Figure 7.10 gives the sequence of basic rules used in the simulation of a composite imperative rule.

7.4 The Cachet Cache Coherence Protocol

In this section, we define the Cachet protocol that employs basic messages only. To ensure liveness, we introduce two basic directive messages, $\mathrm{DownReq}_{\mathrm{mw}}$ and $\mathrm{DownReq}_{\mathrm{wb}}$. Whenever necessary, the memory can send a $\mathrm{DownReq}_{\mathrm{mw}}$ message to downgrade a cache cell from Migratory to WP, or a $\mathrm{DownReq}_{\mathrm{wb}}$ message to downgrade a cache cell from WP to Base. Some memory states in the imperative rules need to incorporate proper information about outstanding directive messages in the integrated rules. The $\mathrm{T}_{\mathrm{w}}[dir,\epsilon]$ state in the imperative rules is split into $\mathrm{C}_{\mathrm{w}}[dir]$ and $\mathrm{T}_{\mathrm{w}}[dir,\epsilon]$ in the integrated rules, where $\mathrm{T}_{\mathrm{w}}[dir,\epsilon]$ implies that the memory has issued a $\mathrm{DownReq}_{\mathrm{wb}}$ message to cache sites *dir*. The $\mathrm{T}_{\mathrm{m}}[id,\epsilon]$ state in the imperative rules is split into $\mathrm{C}_{\mathrm{m}}[id]$, $\mathrm{T}'_{\mathrm{m}}[id]$ and $\mathrm{T}_{\mathrm{m}}[id,\epsilon]$ in the integrated rules, where $\mathrm{T}'_{\mathrm{m}}[id]$ implies that the memory has issued a $\mathrm{DownReq}_{\mathrm{mw}}$ message to cache site *id*, and $\mathrm{T}_{\mathrm{m}}[id,\epsilon]$ implies that the memory has issued a $\mathrm{DownReq}_{\mathrm{mw}}$ followed by a $\mathrm{DownReq}_{\mathrm{wb}}$ to cache site *id*.

7.4.1 Processor Rules of Cachet

Figure 7.11 gives the processor rules of Cachet. The processor rules include the imperative processor rules, and additional rules to deal with stalled instructions. All the processor rules are mandatory rules. Processor rules marked with 'SF' are strongly fair. When an instruction is retired, it is removed from the processor-to-memory buffer; when an instruction is stalled, it remains in the processor-to-memory buffer.

- For a Loadl or Storel instruction, if the address is cached in the Clean or Dirty state of any protocol, the cache supplies the accessed data or an acknowledgment to retire the instruction. If the address is uncached, the cache sends a CacheReq message to request a cache copy from the memory; the instruction remains stalled until the requested data is received.
- For a Commit instruction, if the address is uncached or cached in the Clean state of any protocol or the Dirty state of Migratory, the cache supplies an acknowledgment to retire the instruction. If the address is cached in the Dirty state of Base, the cache sends a Wb_{b} message to write the data back to the memory. If the address is cached in the Dirty state of WP, the cache sends a $\mathrm{Down}_{\mathrm{wb}}$ message followed by a Wb_{b} message to the memory.
- For a Reconcile instruction, if the address is uncached or cached in the Clean state of WP or Migratory or the Dirty state of any protocol, the cache supplies an acknowledgment to retire the instruction. If the address is cached in the Clean state of Base, the cache purges the cache cell to allow the instruction to complete.

7.4.2 Cache Engine Rules of Cachet

Figure 7.12 defines the cache engine rules of Cachet. The cache engine rules contain voluntary rules and mandatory rules. When a cache engine receives a message, it removes the message from the incoming queue. No message needs to be stalled at the cache side. The cache engine

Mandatory Processor Rules					
Instruction	**Cstate**	**Action**	**Next Cstate**		
Loadl(a)	Cell(a,v,Clean$_{\rm b}$)	*retire*	Cell(a,v,Clean$_{\rm b}$)	P1	SF
	Cell(a,v,Dirty$_{\rm b}$)	*retire*	Cell(a,v,Dirty$_{\rm b}$)	P2	SF
	Cell(a,v,Clean$_{\rm w}$)	*retire*	Cell(a,v,Clean$_{\rm w}$)	P3	SF
	Cell(a,v,Dirty$_{\rm w}$)	*retire*	Cell(a,v,Dirty$_{\rm w}$)	P4	SF
	Cell(a,v,Clean$_{\rm m}$)	*retire*	Cell(a,v,Clean$_{\rm m}$)	P5	SF
	Cell(a,v,Dirty$_{\rm m}$)	*retire*	Cell(a,v,Dirty$_{\rm m}$)	P6	SF
	Cell(a,v,WbPending)	*stall*	Cell(a,v,WbPending)	P7	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P8	
	$a \notin$ *cache*	*stall*, $\langle$CacheReq,$a\rangle \rightarrow$ H	Cell(a,-,CachePending)	P9	SF
Storel(a,v)	Cell(a,-,Clean$_{\rm b}$)	*retire*	Cell(a,v,Dirty$_{\rm b}$)	P10	SF
	Cell(a,-,Dirty$_{\rm b}$)	*retire*	Cell(a,v,Dirty$_{\rm b}$)	P11	SF
	Cell(a,-,Clean$_{\rm w}$)	*retire*	Cell(a,v,Dirty$_{\rm w}$)	P12	SF
	Cell(a,-,Dirty$_{\rm w}$)	*retire*	Cell(a,v,Dirty$_{\rm w}$)	P13	SF
	Cell(a,-,Clean$_{\rm m}$)	*retire*	Cell(a,v,Dirty$_{\rm m}$)	P14	SF
	Cell(a,-,Dirty$_{\rm m}$)	*retire*	Cell(a,v,Dirty$_{\rm m}$)	P15	SF
	Cell(a,v_1,WbPending)	*stall*	Cell(a,v_1,WbPending)	P16	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P17	
	$a \notin$ *cache*	*stall*, $\langle$CacheReq,$a\rangle \rightarrow$ H	Cell(a,-,CachePending)	P18	SF
Commit(a)	Cell(a,v,Clean$_{\rm b}$)	*retire*	Cell(a,v,Clean$_{\rm b}$)	P19	SF
	Cell(a,v,Dirty$_{\rm b}$)	*stall*, $\langle$Wb$_{\rm b}$,a,$v\rangle \rightarrow$ H	Cell(a,v,WbPending)	P20	SF
	Cell(a,v,Clean$_{\rm w}$)	*retire*	Cell(a,v,Clean$_{\rm w}$)	P21	SF
	Cell(a,v,Dirty$_{\rm w}$)	*stall*, $\langle$Down$_{\rm wb}$,$a\rangle \rightarrow$ H $\langle$Wb$_{\rm b}$,a,$v\rangle \rightarrow$ H	Cell(a,v,WbPending)	P22	SF
	Cell(a,v,Clean$_{\rm m}$)	*retire*	Cell(a,v,Clean$_{\rm m}$)	P23	SF
	Cell(a,v,Dirty$_{\rm m}$)	*retire*	Cell(a,v,Dirty$_{\rm m}$)	P24	SF
	Cell(a,v,WbPending)	*stall*	Cell(a,v,WbPending)	P25	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P26	
	$a \notin$ *cache*	*retire*	$a \notin$ *cache*	P27	SF
Reconcile(a)	Cell(a,-,Clean$_{\rm b}$)	*retire*	$a \notin$ *cache*	P28	SF
	Cell(a,v,Dirty$_{\rm b}$)	*retire*	Cell(a,v,Dirty$_{\rm b}$)	P29	SF
	Cell(a,v,Clean$_{\rm w}$)	*retire*	Cell(a,v,Clean$_{\rm w}$)	P30	SF
	Cell(a,v,Dirty$_{\rm w}$)	*retire*	Cell(a,v,Dirty$_{\rm w}$)	P31	SF
	Cell(a,v,Clean$_{\rm m}$)	*retire*	Cell(a,v,Clean$_{\rm m}$)	P32	SF
	Cell(a,v,Dirty$_{\rm m}$)	*retire*	Cell(a,v,Dirty$_{\rm m}$)	P33	SF
	Cell(a,v,WbPending)	*stall*	Cell(a,v,WbPending)	P34	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	P35	
	$a \notin$ *cache*	*retire*	$a \notin$ *cache*	P36	SF

Figure 7.11: Processor Rules of Cachet

rules include the imperative cache engine rules, and additional rules to deal with downgrade requests from the memory. A cache processes a downgrade request as follows:

- When a cache receives a WP-to-Base downgrade request, if the address is cached under WP, the cache downgrades the cell to Base, and sends a Down$_{\rm wb}$ message to the memory (Rules MC18 and MC19). However, if the address is cached under Base, or cached in the WbPending or CachePending state, or uncached, the cache simply discards the request (Rules MC16, MC17, MC20, MC21 and MC22). This is because the cache has already downgraded the cell before the downgrade request is received.
- When a cache receives a Migratory-to-WP downgrade request, if the address is cached under Migratory, the cache downgrades the cell to WP, and sends a Down$_{\rm mw}$ or DownV$_{\rm mw}$ message to the memory (Rules MC27 and MC28). However, if the address is cached under Base or WP, or cached in the WbPending or CachePending state, or uncached, the cache simply discards the request (Rules MC23, MC24, MC25, MC26, MC29, MC30 and MC31).

Voluntary C-engine Rules

	Cstate	Action	Next Cstate	
	Cell(a,-,Clean$_{\text{b}}$)		$a \notin$ *cache*	VC1
	Cell(a,v,Dirty$_{\text{b}}$)	$\langle$Wb$_{\text{b}}$,a,$v\rangle \rightarrow$ H	Cell(a,v,WbPending)	VC2
	Cell(a,v,Clean$_{\text{w}}$)	$\langle$Down$_{\text{wb}}$,$a\rangle \rightarrow$ H	Cell(a,v,Clean$_{\text{b}}$)	VC3
	Cell(a,v,Dirty$_{\text{w}}$)	$\langle$Down$_{\text{wb}}$,$a\rangle \rightarrow$ H	Cell(a,v,Dirty$_{\text{b}}$)	VC4
	Cell(a,v,Clean$_{\text{m}}$)	$\langle$Down$_{\text{mw}}$,$a\rangle \rightarrow$ H	Cell(a,v,Clean$_{\text{w}}$)	VC5
	Cell(a,v,Dirty$_{\text{m}}$)	$\langle$DownV$_{\text{mw}}$,a,$v\rangle \rightarrow$ H	Cell(a,v,Clean$_{\text{w}}$)	VC6
	$a \notin$ *cache*	$\langle$CacheReq,$a\rangle \rightarrow$ H	Cell(a,-,CachePending)	VC7

Mandatory C-engine Rules

Msg from H	Cstate	Action	Next Cstate	
$\langle$Cache$_{\text{b}}$,a,$v\rangle$	Cell(a,-,CachePending)		Cell(a,v,Clean$_{\text{b}}$)	MC1
$\langle$Cache$_{\text{w}}$,a,$v\rangle$	Cell(a,-,Clean$_{\text{b}}$)		Cell(a,v,Clean$_{\text{w}}$)	MC2
	Cell(a,v_1,Dirty$_{\text{b}}$)		Cell(a,v_1,Dirty$_{\text{w}}$)	MC3
	Cell(a,v_1,WbPending)		Cell(a,v_1,WbPending)	MC4
	Cell(a,-,CachePending)		Cell(a,v,Clean$_{\text{w}}$)	MC5
	$a \notin$ *cache*		Cell(a,v,Clean$_{\text{w}}$)	MC6
$\langle$Up$_{\text{wm}}$,$a\rangle$	Cell(a,v,Clean$_{\text{b}}$)		Cell(a,v,Clean$_{\text{b}}$)	MC7
	Cell(a,v,Dirty$_{\text{b}}$)		Cell(a,v,Dirty$_{\text{b}}$)	MC8
	Cell(a,v,Clean$_{\text{w}}$)		Cell(a,v,Clean$_{\text{m}}$)	MC9
	Cell(a,v,Dirty$_{\text{w}}$)		Cell(a,v,Dirty$_{\text{m}}$)	MC10
	Cell(a,v,WbPending)		Cell(a,v,WbPending)	MC11
	Cell(a,-,CachePending)		Cell(a,-,CachePending)	MC12
	$a \notin$ *cache*		$a \notin$ *cache*	MC13
$\langle$WbAck$_{\text{b}}$,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean$_{\text{b}}$)	MC14
$\langle$WbAck$_{\text{w}}$,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean$_{\text{w}}$)	MC15
$\langle$DownReq$_{\text{wb}}$,$a\rangle$	Cell(a,v,Clean$_{\text{b}}$)		Cell(a,v,Clean$_{\text{b}}$)	MC16
	Cell(a,v,Dirty$_{\text{b}}$)		Cell(a,v,Dirty$_{\text{b}}$)	MC17
	Cell(a,v,Clean$_{\text{w}}$)	$\langle$Down$_{\text{wb}}$,$a\rangle \rightarrow$ H	Cell(a,v,Clean$_{\text{b}}$)	MC18
	Cell(a,v,Dirty$_{\text{w}}$)	$\langle$Down$_{\text{wb}}$,$a\rangle \rightarrow$ H	Cell(a,v,Dirty$_{\text{b}}$)	MC19
	Cell(a,v,WbPending)		Cell(a,v,WbPending)	MC20
	Cell(a,-,CachePending)		Cell(a,-,CachePending)	MC21
	$a \notin$ *cache*		$a \notin$ *cache*	MC22
$\langle$DownReq$_{\text{mw}}$,$a\rangle$	Cell(a,v,Clean$_{\text{b}}$)		Cell(a,v,Clean$_{\text{b}}$)	MC23
	Cell(a,v,Dirty$_{\text{b}}$)		Cell(a,v,Dirty$_{\text{b}}$)	MC24
	Cell(a,v,Clean$_{\text{w}}$)		Cell(a,v,Clean$_{\text{w}}$)	MC25
	Cell(a,v,Dirty$_{\text{w}}$)		Cell(a,v,Dirty$_{\text{w}}$)	MC26
	Cell(a,v,Clean$_{\text{m}}$)	$\langle$Down$_{\text{mw}}$,$a\rangle \rightarrow$ H	Cell(a,v,Clean$_{\text{w}}$)	MC27
	Cell(a,v,Dirty$_{\text{m}}$)	$\langle$DownV$_{\text{mw}}$,a,$v\rangle \rightarrow$ H	Cell(a,v,Clean$_{\text{w}}$)	MC28
	Cell(a,v,WbPending)		Cell(a,v,WbPending)	MC29
	Cell(a,-,CachePending)		Cell(a,-,CachePending)	MC30
	$a \notin$ *cache*		$a \notin$ *cache*	MC31

Figure 7.12: Cache Engine Rules of Cachet

This is because the cache has already downgraded the cell before the downgrade request is received.

7.4.3 Memory Engine Rules of Cachet

Figure 7.13 defines the memory engine rules of Cachet. The memory engine rules consist of voluntary rules and mandatory rules. An incoming message can be processed or stalled when it is received. When a message is processed, it is removed from the incoming queue; when a message is stalled, it remains in the incoming queue for later processing. Note that only CacheReq messages can be stalled; a stalled message cannot block following messages from being processed.

Voluntary M-engine Rules				
	Mstate	**Action**	**Next Mstate**	
	$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])\ (id \notin dir)$	$\langle \mathrm{Cache_w},a,v\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{C_w}[id \mid dir])$	VM1
	$\mathrm{Cell}(a,v,\mathrm{C_w}[id])$	$\langle \mathrm{Up_{wm}},a\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$	VM2
	$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])$	$\langle \mathrm{DownReq_{wb}},a\rangle \rightarrow dir$	$\mathrm{Cell}(a,v,\mathrm{T_w}[dir,\epsilon])$	VM3
	$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$	$\langle \mathrm{DownReq_{mw}},a\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{T}'_\mathrm{m}[id])$	VM4
	$\mathrm{Cell}(a,v,\mathrm{T}'_\mathrm{m}[id])$	$\langle \mathrm{DownReq_{wb}},a\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{T_m}[id,\epsilon])$	VM5

Mandatory M-engine Rules				
Msg from id	**Mstate**	**Action**	**Next Mstate**	
$\langle \mathrm{CacheReq},a\rangle$	$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])\ (id \notin dir)$	$\langle \mathrm{Cache_b},a,v\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])$	MM1
	$\mathrm{Cell}(a,v,\mathrm{T_w}[dir,sm])\ (id \notin dir)$	*stall message*	$\mathrm{Cell}(a,v,\mathrm{T_w}[dir,sm])$	MM2
	$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])\ (id \in dir)$		$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])$	MM3
	$\mathrm{Cell}(a,v,\mathrm{T_w}[dir,sm])\ (id \in dir)$		$\mathrm{Cell}(a,v,\mathrm{T_w}[dir,sm])$	MM4
	$\mathrm{Cell}(a,v,\mathrm{C_m}[id_1])\ (id \neq id_1)$	*stall message* $\langle \mathrm{DownReq_{mw}},a\rangle \rightarrow id_1$	$\mathrm{Cell}(a,v,\mathrm{T}'_\mathrm{m}[id_1])$	MM5
	$\mathrm{Cell}(a,v,\mathrm{T}'_\mathrm{m}[id_1])\ (id \neq id_1)$	*stall message*	$\mathrm{Cell}(a,v,\mathrm{T}'_\mathrm{m}[id_1])$	MM6
	$\mathrm{Cell}(a,v,\mathrm{T_m}[id_1,sm])\ (id \neq id_1)$	*stall message*	$\mathrm{Cell}(a,v,\mathrm{T_m}[id_1,sm])$	MM7
	$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$	MM8
	$\mathrm{Cell}(a,v,\mathrm{T}'_\mathrm{m}[id])$		$\mathrm{Cell}(a,v,\mathrm{T}'_\mathrm{m}[id])$	MM9
	$\mathrm{Cell}(a,v,\mathrm{T_m}[id,sm])$		$\mathrm{Cell}(a,v,\mathrm{T_m}[id,sm])$	MM10
$\langle \mathrm{Wb_b},a,v\rangle$	$\mathrm{Cell}(a,v_1,\mathrm{C_w}[dir])\ (id \notin dir)$	$\langle \mathrm{DownReq_{wb}},a\rangle \rightarrow dir$	$\mathrm{Cell}(a,v_1,\mathrm{T_w}[dir,(id,v)])$	MM11
	$\mathrm{Cell}(a,v_1,\mathrm{T_w}[dir,sm])\ (id \notin dir)$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[dir,(id,v) \mid sm])$	MM12
	$\mathrm{Cell}(a,v_1,\mathrm{C_w}[id \mid dir])$	$\langle \mathrm{DownReq_{wb}},a\rangle \rightarrow dir$	$\mathrm{Cell}(a,v_1,\mathrm{T_w}[dir,(id,v)])$	MM13
	$\mathrm{Cell}(a,v_1,\mathrm{T_w}[id \mid dir,sm])$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[dir,(id,v) \mid sm])$	MM14
	$\mathrm{Cell}(a,v_1,\mathrm{C_m}[id_1])\ (id \neq id_1)$	$\langle \mathrm{DownReq_{mw}},a\rangle \rightarrow id_1$ $\langle \mathrm{DownReq_{wb}},a\rangle \rightarrow id_1$	$\mathrm{Cell}(a,v_1,\mathrm{T_m}[id_1,(id,v)])$	MM15
	$\mathrm{Cell}(a,v_1,\mathrm{T}'_\mathrm{m}[id_1])\ (id \neq id_1)$	$\langle \mathrm{Down_{wb}},a\rangle \rightarrow id_1$	$\mathrm{Cell}(a,v_1,\mathrm{T_m}[id_1,(id,v)])$	MM16
	$\mathrm{Cell}(a,v_1,\mathrm{T_m}[id_1,sm])\ (id \neq id_1)$		$\mathrm{Cell}(a,v_1,\mathrm{T_m}[id_1,(id,v) \mid sm])$	MM17
	$\mathrm{Cell}(a,v_1,\mathrm{C_m}[id])$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[\epsilon,(id,v)])$	MM18
	$\mathrm{Cell}(a,v_1,\mathrm{T}'_\mathrm{m}[id])$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[\epsilon,(id,v)])$	MM19
	$\mathrm{Cell}(a,v_1,\mathrm{T_m}[id,sm])$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[\epsilon,(id,v) \mid sm])$	MM20
$\langle \mathrm{Down_{wb}},a\rangle$	$\mathrm{Cell}(a,v,\mathrm{C_w}[id \mid dir])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])$	MM21
	$\mathrm{Cell}(a,v,\mathrm{T_w}[id \mid dir,sm])$		$\mathrm{Cell}(a,v,\mathrm{T_w}[dir,sm])$	MM22
	$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[\epsilon])$	MM23
	$\mathrm{Cell}(a,v,\mathrm{T}'_\mathrm{m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[\epsilon])$	MM24
	$\mathrm{Cell}(a,v,\mathrm{T_m}[id,sm])$		$\mathrm{Cell}(a,v,\mathrm{T_w}[\epsilon,sm])$	MM25
$\langle \mathrm{Down_{mw}},a\rangle$	$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[id])$	MM26
	$\mathrm{Cell}(a,v,\mathrm{T}'_\mathrm{m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[id])$	MM27
	$\mathrm{Cell}(a,v,\mathrm{T_m}[id,sm])$		$\mathrm{Cell}(a,v,\mathrm{T_w}[id,sm])$	MM28
$\langle \mathrm{DownV_{mw}},a,v\rangle$	$\mathrm{Cell}(a,\text{-},\mathrm{C_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[id])$	MM29
	$\mathrm{Cell}(a,\text{-},\mathrm{T}'_\mathrm{m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[id])$	MM30
	$\mathrm{Cell}(a,\text{-},\mathrm{T_m}[id,sm])$		$\mathrm{Cell}(a,v,\mathrm{T_w}[id,sm])$	MM31
	$\mathrm{Cell}(a,\text{-},\mathrm{T_w}[\epsilon,(id,v) \mid sm])$	$\langle \mathrm{WbAck_b},a\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{T_w}[\epsilon,sm])$	MM32
	$\mathrm{Cell}(a,\text{-},\mathrm{T_w}[\epsilon,(id,v)])$	$\langle \mathrm{WbAck_w},a\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{C_w}[id])$	MM33
	$\mathrm{Cell}(a,v,\mathrm{T_w}[\epsilon,\epsilon])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[\epsilon])$	MM34

Figure 7.13: Memory Engine Rules of Cachet (Rule MM1 is strongly fair)

The memory processes an incoming CacheReq message from cache site *id* as follows:

- If the memory state is $\mathrm{C_w}[dir]$ $(id \notin dir)$, the memory sends a $\mathrm{Cache_b}$ message to supply a Base copy to the cache (Rule MM1). An alternative treatment is that the memory sends a $\mathrm{Cache_w}$ message to supply a WP copy (Rule VM1) and discards the cache request (Rule MM3).
- If the memory state is $\mathrm{C_m}[id_1]$ $(id \neq id_1)$, the memory sends a $\mathrm{DownReq_{mw}}$ message to downgrade the Migratory cell at cache id_1 (Rule MM5). The cache request remains stalled until the downgrade request is acknowledged.
- If the memory state shows that the address is already cached in the cache, the memory discards the cache request (Rules MM3, MM4, MM8, MM9 and MM10). This can happen because the memory can voluntarily supply a cache copy to a cache.
- If the memory state is transient and shows that address is not cached in the cache, the message is stalled for later processing (Rules MM2, MM6 and MM7). The stalled message cannot be processed before the memory state becomes stable.

It is worth noting that Rule MM1 is strongly fair. This guarantees that a cache request cannot be blocked forever while cache requests from other cache sites on the same address are processed repeatedly. Such fairness is critical to ensure that each cache request will be serviced eventually.

The memory processes an incoming $\mathrm{Wb_b}$ message from cache site *id* as follows:

- If the memory state is $\mathrm{C_w}[dir]$ or $\mathrm{T_w}[dir,sm]$ $(id \notin dir)$, the memory suspends the writeback message (Rules MM11 and MM12). For the $\mathrm{C_w}[dir]$ state, the memory multicasts a $\mathrm{DownReq_{wb}}$ message to cache sites *dir* to downgrade the cache cells from WP to Base.
- If the memory state is $\mathrm{C_w}[id|dir]$ or $\mathrm{T_w}[id|dir,sm]$, the memory suspends the writeback message and removes the cache identifier from the directory (Rules MM13 and MM14). For the $\mathrm{C_w}[id|dir]$ state, the memory multicasts a $\mathrm{DownReq_{wb}}$ message to cache sites *dir*.
- If the memory state is $\mathrm{C_m}[id_1]$, $\mathrm{T'_m}[id_1]$ or $\mathrm{T_m}[id_1,sm]$ $(id \neq id_1)$, the memory suspends the writeback message (Rules MM15, MM16 and MM17). For the $\mathrm{C_m}[id_1]$ state, the memory sends a $\mathrm{DownReq_{mw}}$ message followed by a $\mathrm{DownReq_{wb}}$ message to cache site id_1; for the $\mathrm{T'_m}[id_1]$ state, the memory sends a $\mathrm{DownReq_{wb}}$ message to cache site id_1.
- If the memory state is $\mathrm{C_m}[id]$, $\mathrm{T'_m}[id]$ or $\mathrm{T_m}[id,sm]$, the memory suspends the writeback message and updates the memory state to indicate that the address is uncached in any cache site (Rules MM18, MM19 and MM20). This can happen because the memory can voluntarily send a $\mathrm{Cache_w}$ message followed by an $\mathrm{Up_{wm}}$ message to a cache before it receives the writeback message.

Suspended writeback messages can be resumed when the memory state shows that the address is not cached in any cache. When a writeback message is resumed, the memory sends a $\mathrm{WbAck_b}$ message to the cache to acknowledge that the writeback operation has been performed

(Rule MM32). The memory can choose to acknowledge the last writeback with a $\mathrm{WbAck_w}$ message to allow the cache to retain a WP copy (Rule MM33).

The memory processes an incoming $\mathrm{Down_{wb}}$, $\mathrm{Down_{mw}}$ or $\mathrm{DownV_{mw}}$ message as follows:

- When the memory receives a $\mathrm{Down_{wb}}$ message, if the memory state shows that the cache contains a WP copy for the address, the memory removes the cache identifier from the corresponding directory (Rules MM21 and MM22). If the memory state shows that the cache contains a Migratory copy for the address, the memory updates the memory state to indicate that the address is no longer cached in any cache (Rules MM23, MM24 and MM25). This can happen because the memory can voluntarily send an upgrade message to upgrade a cache cell from WP to Migratory, while the cache has downgraded the cache cell from WP to Base. The downgrade operation has higher priority than the upgrade operation.
- When the memory receives a $\mathrm{Down_{mw}}$ message, it sets the memory state to indicate that the cache contains a WP copy for the address (Rules MM26, MM27 and MM28).
- When the memory receives a $\mathrm{DownV_{mw}}$ message, it updates the memory value and sets the memory state to indicate that the cache contains a WP copy for the address (Rules MM29, MM30 and MM31).

In addition, the $\mathrm{T_w}[\epsilon,\epsilon]$ state can be converted to $\mathrm{C}[\epsilon]$. This is necessary to ensure that a transient memory state will become a stable memory state eventually so that stalled cache requests can be serviced.

Voluntary Rules There are five voluntary rules that allow the memory to supply a cache copy to a cache, to upgrade a cache cell or to downgrade a cache cell.

- If the memory state is $\mathrm{C_w}[dir]$, the memory can send a $\mathrm{Cache_w}$ message to supply a WP copy to cache site id, where $id \notin dir$.
- If the memory state is $\mathrm{C_w}[id]$, the memory can send an $\mathrm{Up_{wm}}$ message to cache site id to upgrade the cache cell from WP to Migratory.
- If the memory state is $\mathrm{C_w}[dir]$, the memory can multicast a $\mathrm{DownReq_{wb}}$ message to cache sites dir to downgrade the cache cells from WP to Base.
- If the memory state is $\mathrm{C_m}[id]$, the memory can send a $\mathrm{DownReq_{mw}}$ message to cache site id to downgrade the cache cell from Migratory to WP.
- If the memory state is $\mathrm{T'_m}[id]$, the memory can send a $\mathrm{DownReq_{wb}}$ message to cache site id to downgrade the cache cell from WP to Base.

The memory state can be imprecise since it maintains no information about the existence of Base cells. In addition, when the memory state shows that a cache contains a WP or Migratory cell for an address, it is possible that the cache cell has already been downgraded.

The inaccuracy of memory states can cause unexpected cases that must be dealt with properly. Some scenarios are discussed as follows:

- Simultaneous Cache$_{\mathrm{w}}$ and Wb$_{\mathrm{b}}$. Suppose initially the address is cached in the Dirty state of Base in a cache, while the memory state shows that the address is uncached in the cache. The memory sends a Cache$_{\mathrm{w}}$ message to supply a WP copy to the cache, while the cache sends a Wb$_{\mathrm{b}}$ message to write the dirty copy back to the memory. The Cache$_{\mathrm{w}}$ message will be discarded when it is received (Rule MC4).
- Simultaneous Up$_{\mathrm{wm}}$ and Down$_{\mathrm{mw}}$. Suppose initially the address is cached in the Clean or Dirty state of WP, while the memory state shows that the address is exclusively cached under WP in the cache. The memory sends an Up$_{\mathrm{wm}}$ message to upgrade the cache cell from WP to Migratory, while the cache downgrades the cell from WP to Base and sends a Down$_{\mathrm{wb}}$ message to the memory. The Up$_{\mathrm{wm}}$ message will be discarded when it is received (Rules MC7 and MC8).
- Simultaneous DownReq$_{\mathrm{wb}}$ and Down$_{\mathrm{wb}}$. Suppose initially the address is cached in the Clean or Dirty state of WP. The memory sends a DownReq$_{\mathrm{wb}}$ message to the cache, while the cache downgrades the cell from WP to Base before it receives the request. The DownReq$_{\mathrm{wb}}$ message will be discarded when it is received (Rules MC16 and MC17).

Buffer Management When a cache request is stalled, it should not block other messages in the incoming queue from being processed. This can be achieved via proper buffer management to allow an incoming message to overtake a cache request as long as the two messages are from different cache sites or have different addresses. This can be characterized as follows:

Cachet's buffer management:

$$\begin{array}{lll}
msg_1 \odot msg_2 & \equiv & msg_2 \odot msg_1 \\
\quad \textit{if} & & (\mathrm{Cmd}(msg_1) = \mathrm{CacheReq} \;\vee\; \mathrm{Cmd}(msg_2) = \mathrm{CacheReq}) \;\wedge \\
 & & (\mathrm{Src}(msg_1) \neq \mathrm{Src}(msg_2) \;\vee\; \mathrm{Addr}(msg_1) \neq \mathrm{Addr}(msg_2))
\end{array}$$

7.4.4 Derivation of Cachet from Imperative and Directive Rules

The Cachet rules can be derived from the imperative and directive rules. A directive rule can be used to generate or discard a directive message; it cannot modify any system state that may affect the soundness of the system. There are four basic directive rules that involve the basic directive messages.

- Send-DownReq$_{\mathrm{wb}}$: The memory sends a DownReq$_{\mathrm{wb}}$ message to a cache.
- Send-DownReq$_{\mathrm{mw}}$: The memory sends a DownReq$_{\mathrm{mw}}$ message to a cache.
- Receive-DownReq$_{\mathrm{wb}}$: The cache discards an incoming DownReq$_{\mathrm{wb}}$ message.
- Receive-DownReq$_{\mathrm{mw}}$: The cache discards an incoming DownReq$_{\mathrm{mw}}$ message.

Processor Rule of Cachet	Deriving Imperative & Directive Rules
P1	IP1
P2	IP2
P3	IP3
P4	IP4
P5	IP5
P6	IP6
P7	ϵ
P8	ϵ
P9	IC7
P10	IP7
P11	IP8
P12	IP9
P13	IP10
P14	IP11
P15	IP12
P16	ϵ
P17	ϵ
P18	IC7
P19	IP13
P20	IC2
P21	IP14
P22	IC4 + IC2
P23	IP15
P24	IP16
P25	ϵ
P26	ϵ
P27	IP17
P28	IC1 + IP23
P29	IP18
P30	IP19
P31	IP20
P32	IP21
P33	IP22
P34	ϵ
P35	ϵ
P36	IP23

Figure 7.14: Derivation of Processor Rules of Cachet

Figures 7.14, 7.15 and 7.16 give the derivations of the processor rules, the cache engine rules and the memory engine rules, respectively. In the derivation, the $\mathrm{C_w}[dir]$ state used in the integrated rules corresponds to the $\mathrm{T_w}[dir,\epsilon]$ state of the imperative rules; the $\mathrm{C_m}[id]$ and $\mathrm{T'_m}[id]$ states used in the integrated rules correspond to the $\mathrm{T_m}[id,\epsilon]$ state of the imperative rules.

7.5 The Composite Rules of Cachet

In this section, we will present some composite rules that involve the composite messages. Since a composite message behaves as a sequence of basic messages, a composite rule can always be simulated by some basic rules. Figure 7.17 defines the composite rules of Cachet.

When a Commit instruction is performed on an address that is cached in the Dirty state of WP, the cache can write the data back to the memory via a $\mathrm{Wb_w}$ message. The instruction remains stalled until the writeback operation is acknowledged.

C-engine Rule of Cachet	Deriving Imperative & Directive Rules
VC1	IC1
VC2	IC2
VC3	IC3
VC4	IC4
VC5	IC5
VC6	IC6
VC7	IC7
MC1	IC8
MC2	IC1 + IC13
MC3	IC10
MC4	IC11
MC5	IC12
MC6	IC13
MC7	IC14
MC8	IC15
MC9	IC16
MC10	IC17
MC11	IC18
MC12	IC19
MC13	IC20
MC14	IC21
MC15	IC22
MC16	Receive-DownReq$_{\text{wb}}$
MC17	Receive-DownReq$_{\text{wb}}$
MC18	Receive-DownReq$_{\text{wb}}$ + IC3
MC19	Receive-DownReq$_{\text{wb}}$ + IC4
MC20	Receive-DownReq$_{\text{wb}}$
MC21	Receive-DownReq$_{\text{wb}}$
MC22	Receive-DownReq$_{\text{wb}}$
MC23	Receive-DownReq$_{\text{mw}}$
MC24	Receive-DownReq$_{\text{mw}}$
MC25	Receive-DownReq$_{\text{mw}}$
MC26	Receive-DownReq$_{\text{mw}}$
MC27	Receive-DownReq$_{\text{mw}}$ + IC5
MC28	Receive-DownReq$_{\text{mw}}$ + IC6
MC29	Receive-DownReq$_{\text{mw}}$
MC30	Receive-DownReq$_{\text{mw}}$
MC31	Receive-DownReq$_{\text{mw}}$

Figure 7.15: Derivation of Cache Engine Rules of Cachet

The composite C-engine rules include the composite imperative C-engine rules given in Figure 7.9, and some additional rules that are needed to process Migratory-to-Base downgrade requests. A Migratory-to-Base message behaves as a DownReq$_{\text{mw}}$ followed by a DownReq$_{\text{wb}}$.

The memory may send a composite message to a cache under certain circumstances. If the memory state is C$_{\text{w}}[\epsilon]$, the memory can send a Migratory message to supply a Migratory copy to a cache site. If the memory state is C$_{\text{m}}[id]$, the memory can send a DownReq$_{\text{mb}}$ message to cache site *id* to downgrade the cache cell from Migratory to Base. In addition, the memory can acknowledge the last resumed writeback operation with a WbAck$_{\text{m}}$ message to allow the cache to retain a Migratory copy. When the memory receives a composite message, it processes the message as a sequence of basic messages. The memory treats a Wb$_{\text{w}}$ message as a Down$_{\text{wb}}$ followed by a Wb$_{\text{b}}$, a Down$_{\text{mb}}$ message as a Down$_{\text{mw}}$ followed by a Down$_{\text{wb}}$, and a DownV$_{\text{mb}}$ message as a DownV$_{\text{mw}}$ followed by a Down$_{\text{wb}}$.

M-engine Rule of Cachet	Deriving Imperative & Directive Rules
VM1	IM1
VM2	IM2
VM3	Send-DownReq$_{\rm wb}^*$
VM4	Send-DownReq$_{\rm mw}$
VM5	Send-DownReq$_{\rm wb}$
MM1	IM3
MM2	ϵ
MM3	IM4
MM4	IM4
MM5	Send-DownReq$_{\rm mw}$
MM6	ϵ
MM7	ϵ
MM8	IM5
MM9	IM5
MM10	IM5
MM11	IM6 + Send-DownReq$_{\rm wb}^*$
MM12	IM6
MM13	IM7 + Send-DownReq$_{\rm wb}^*$
MM14	IM7
MM15	IM8 + Send-DownReq$_{\rm mw}$ + Send-DownReq$_{\rm wb}$
MM16	IM8 + Send-DownReq$_{\rm wb}$
MM17	IM8
MM18	IM9
MM19	IM9
MM20	IM9
MM21	IM10
MM22	IM10
MM23	IM11
MM24	IM11
MM25	IM11
MM26	IM12
MM27	IM12
MM28	IM12
MM29	IM13
MM30	IM13
MM31	IM13
MM32	IM14
MM33	IM15
MM34	ϵ

Figure 7.16: Derivation of Memory Engine Rules of Cachet

Figure 7.18 gives the basic rules used in the simulation of each composite rule.

The Cachet Protocol Appendix A gives the specification of the complete Cachet protocol, which contains the basic rules defined in Figures 7.11, 7.12 and 7.13, and the composite rules defined in Figure 7.17. The Cachet protocol is an adaptive cache coherence protocol, although for pedagogic reason it has been been presented as an integration of several micro-protocols. One can also think Cachet as a family of protocols because of the presence of voluntary rules that can be invoked without the execution of an instruction or the receipt of a message. The existence of voluntary rules provides enormous extensibility in the sense that various heuristic messages and states can be employed to invoke these rules.

As an example of how the adaptivity can be exploited, consider a DSM system with limited directory space. When the memory receives a cache request, it can respond under Base or WP.

Composite Mandatory Processor Rules

Instruction	Cstate	Action	Next Cstate	
Commit(a)	Cell(a,v,Dirty$_{\text{w}}$)	*stall*, $\langle \text{Wb}_{\text{w}},a,v\rangle \rightarrow \text{H}$	Cell(a,v,WbPending)	CP1

Composite Voluntary C-engine Rules

	Cstate	Action	Next Cstate	
	Cell(a,v,Dirty$_{\text{w}}$)	$\langle \text{Wb}_{\text{w}},a,v\rangle \rightarrow \text{H}$	Cell(a,v,WbPending)	CVC1
	Cell(a,v,Clean$_{\text{m}}$)	$\langle \text{Down}_{\text{mb}},a\rangle \rightarrow \text{H}$	Cell(a,v,Clean$_{\text{b}}$)	CVC2
	Cell(a,v,Dirty$_{\text{m}}$)	$\langle \text{DownV}_{\text{mb}},a,v\rangle \rightarrow \text{H}$	Cell(a,v,Clean$_{\text{b}}$)	CVC3

Composite Mandatory C-engine Rules

$\langle \text{Cache}_{\text{m}},a,v\rangle$	Cell(a,-,Clean$_{\text{b}}$)		Cell(a,v,Clean$_{\text{m}}$)	CMC1
	Cell(a,v_1,Dirty$_{\text{b}}$)		Cell(a,v_1,Dirty$_{\text{m}}$)	CMC2
	Cell(a,v_1,WbPending)		Cell(a,v_1,WbPending)	CMC3
	Cell(a,-,CachePending)		Cell(a,v,Clean$_{\text{m}}$)	CMC4
	$a \notin$ *cache*		Cell(a,v,Clean$_{\text{m}}$)	CMC5
$\langle \text{WbAck}_{\text{m}},a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean$_{\text{m}}$)	CMC6
$\langle \text{DownReq}_{\text{mb}},a\rangle$	Cell(a,v,Clean$_{\text{b}}$)		Cell(a,v,Clean$_{\text{b}}$)	CMC7
	Cell(a,v,Dirty$_{\text{b}}$)		Cell(a,v,Dirty$_{\text{b}}$)	CMC8
	Cell(a,v,Clean$_{\text{w}}$)	$\langle \text{Down}_{\text{wb}},a\rangle \rightarrow \text{H}$	Cell(a,v,Clean$_{\text{b}}$)	CMC9
	Cell(a,v,Dirty$_{\text{w}}$)	$\langle \text{Down}_{\text{wb}},a\rangle \rightarrow \text{H}$	Cell(a,v,Dirty$_{\text{b}}$)	CMC10
	Cell(a,v,Clean$_{\text{m}}$)	$\langle \text{Down}_{\text{mb}},a\rangle \rightarrow \text{H}$	Cell(a,v,Clean$_{\text{b}}$)	CMC11
	Cell(a,v,Dirty$_{\text{m}}$)	$\langle \text{DownV}_{\text{mb}},a,v\rangle \rightarrow \text{H}$	Cell(a,v,Clean$_{\text{b}}$)	CMC12
	Cell(a,v,WbPending)		Cell(a,v,WbPending)	CMC13
	Cell(a,-,CachePending)		Cell(a,-,CachePending)	CMC14
	$a \notin$ *cache*		$a \notin$ *cache*	CMC15

Composite Voluntary M-engine Rules

	Mstate	Action	Next Mstate	
	Cell(a,v,C$_{\text{w}}[\epsilon]$)	$\langle \text{Cache}_{\text{m}},a,v\rangle \rightarrow \mathit{id}$	Cell(a,v,C$_{\text{m}}[\mathit{id}]$)	CVM1
	Cell(a,v,C$_{\text{m}}[\mathit{id}]$)	$\langle \text{DownReq}_{\text{mb}},a\rangle \rightarrow \mathit{id}$	Cell(a,v,T$_{\text{m}}[\mathit{id},\epsilon]$)	CVM2

Composite Mandatory M-engine Rules

$\langle \text{Wb}_{\text{w}},a,v\rangle$	Cell(a,v_1,C$_{\text{w}}[\mathit{id} \mid \mathit{dir}]$)	$\langle \text{DownReq}_{\text{wb}},a\rangle \rightarrow \mathit{dir}$	Cell(a,v_1,T$_{\text{w}}[\mathit{dir},(\mathit{id},v)]$)	CMM1
	Cell(a,v_1,T$_{\text{w}}[\mathit{id} \mid \mathit{dir},\mathit{sm}]$)		Cell(a,v_1,T$_{\text{w}}[\mathit{dir},(\mathit{id},v) \mid \mathit{sm}]$)	CMM2
	Cell(a,v_1,C$_{\text{m}}[\mathit{id}]$)		Cell(a,v_1,T$_{\text{w}}[\epsilon,(\mathit{id},v)]$)	CMM3
	Cell(a,v_1,T$'_{\text{m}}[\mathit{id}]$)		Cell(a,v_1,T$_{\text{w}}[\epsilon,(\mathit{id},v)]$)	CMM4
	Cell(a,v_1,T$_{\text{m}}[\mathit{id},\mathit{sm}]$)		Cell(a,v_1,T$_{\text{w}}[\epsilon,(\mathit{id},v) \mid \mathit{sm}]$)	CMM5
$\langle \text{Down}_{\text{mb}},a\rangle$	Cell(a,v,C$_{\text{m}}[\mathit{id}]$)		Cell(a,v,C$_{\text{w}}[\epsilon]$)	CMM6
	Cell(a,v,T$'_{\text{m}}[\mathit{id}]$)		Cell(a,v,C$_{\text{w}}[\epsilon]$)	CMM7
	Cell(a,v,T$_{\text{m}}[\mathit{id},\mathit{sm}]$)		Cell(a,v,T$_{\text{w}}[\epsilon,\mathit{sm}]$)	CMM8
$\langle \text{DownV}_{\text{mb}},a,v\rangle$	Cell(a,-,C$_{\text{m}}[\mathit{id}]$)		Cell(a,v,C$_{\text{w}}[\epsilon]$)	CMM9
	Cell(a,-,T$'_{\text{m}}[\mathit{id}]$)		Cell(a,v,C$_{\text{w}}[\epsilon]$)	CMM10
	Cell(a,-,T$_{\text{m}}[\mathit{id},\mathit{sm}]$)		Cell(a,v,T$_{\text{w}}[\epsilon,\mathit{sm}]$)	CMM11
	Cell(a,-,T$_{\text{w}}[\epsilon,(\mathit{id},v)]$)	$\langle \text{WbAck}_{\text{m}},a\rangle \rightarrow \mathit{id}$	Cell(a,v,C$_{\text{m}}[\mathit{id}]$)	CMM12

Figure 7.17: Composite Rules of Cachet

Composite Rule	Simulating Rules
CP1	P22
CVC1	VC4 + VC2
CVC2	VC5 + VC3
CVC3	VC6 + VC3
CMC1	MC2 + MC9
CMC2	MC3 + MC10
CMC3	MC4 + MC11
CMC4	MC5 + MC9
CMC5	MC6 + MC9
CMC6	MC15 + MC9
CMC7	MC23 + MC16
CMC8	MC24 + MC17
CMC9	MC25 + MC18
CMC10	MC26 + MC19
CMC11	MC27 + MC18
CMC12	MC28 + MC18
CMC13	MC29 + MC20
CMC14	MC30 + MC21
CMC15	MC31 + MC22
CVM1	VM1 + VM2
CVM2	VM4 + VM5
CMM1	MM21 + MM11
CMM2	MM22 + MM12
CMM3	MM23 + MM11
CMM4	MM24 + MM11
CMM5	MM24 + MM12
CMM6	MM26 + MM21
CMM7	MM27 + MM21
CMM8	MM28 + MM22
CMM9	MM29 + MM21
CMM10	MM30 + MM21
CMM11	MM31 + MM22
CMM12	MM33 + VM2

Figure 7.18: Simulation of Composite Rules of Cachet

One reasonable strategy is to always supply a WP copy except when the directory is full, in which case a Base copy is supplied. Meanwhile, the memory can send a heuristic downgrade request message to a cache that the memory chooses as a potential victim. The intention of the heuristic message is to suggest that some cache cell be downgraded from WP to Base so that the resumed directory space can be used for other WP copies.

It is worth emphasizing that the heuristic request message is just a hint to the cache; the cache may or may not satisfy the request. When the cache receives the heuristic request message, it can invoke the appropriate voluntary rule to downgrade the cache cell, or ignore the heuristic message if it intends to retain the WP cell for later reference. This simple adaptivity will allow an address to be resident in more caches than the number of cache identifier slots in the directory.

Since Cachet implements the CRF memory model, it is automatically a cache coherence protocol that implements all the memory models whose programs can be translated into CRF programs. The translation can be performed statically by the compiler or dynamically by the protocol engines. This implies that different memory models can be applied in different memory regions simultaneously. In a program that assumes release consistency, for example, the memory region used for input and output operations can have the semantics of sequential consistency by employing an appropriate translation scheme for that region.

With both Cachet and CRF specified in Term Rewriting Systems, we can formally prove that the Cachet protocol is a correct implementation of the CRF model and is free from any type of deadlock or livelock. The verification of Cachet follows the same procedure as the verification of the Base, WP and Migratory protocols. To prove the soundness, for example, we define a mapping function from Cachet to CRF, and show that each basic imperative rule of Cachet can be simulated in CRF. The mapping function is based on the notion of drained terms, in which all message queues are empty. There are many ways to drain messages from the network. For example, we can employ backward draining for $\mathrm{Wb_b}$ messages, and forward draining for all other messages. The draining rules include Rules IC8-IC22, IM3-IM5, IM10-IM13, and some additional rules that allow $\mathrm{Wb_b}$ messages to be reclaimed at the cache sites from which they were issued. Furthermore, we can downgrade all Migratory cells to WP cells to ensure that the memory in a drained term always contains the most up-to-date data. This can be achieved by including Rules IC5 and IC6 in the draining rules. The Imperative-&-Directive methodology, together with the classification of basic and composite operations, has significantly simplified the verification by reducing the number of rules that need to be considered.

Coarse-grain Coherence States An implementation can maintain coherence states for cache lines rather than individual cache cells. In modern computer systems, the size of cache lines typically ranges from 32 to 128 bytes. The state of a cache line is a concise representation of the states of the cells in the cache line. The $\mathrm{Clean_w^*}$ state, for example, means that all the cache cells of the cache line are in the $\mathrm{Clean_w}$ state. When the cache line is modified by a Storel instruction, the state becomes $\mathrm{Clean_w^*{\cdot}Dirty_w^+}$, which implies that at least one cache cell of the cache line is in the $\mathrm{Dirty_w}$ state while all other cache cells, if any, are in the $\mathrm{Clean_w}$ state. Coherence actions such as cache, writeback, downgrade and upgrade operations are all performed at the cache line granularity. This ensures that all the cache cells in a cache line employ the same micro-protocol at any time.

Since a cache cell can be modified without the exclusive ownership, there can be multiple writers for the same cache line simultaneously. This can happen even for data-race-free programs because of false sharing. As a result, when a cache line in the $\mathrm{Clean_w^*{\cdot}Dirty_w^+}$ state is written back to the memory, the memory should be able to distinguish between clean and dirty cells because only the data of dirty cells can be used to update the memory. A straightforward solution is to maintain a modification bit for each cache cell. At the writeback time, the modification bits are also sent back so that the memory can tell which cache cells have been modified. By allowing a cache line to be modified without the exclusive ownership, the Cachet protocol not only reduces the latency of write operations, but also alleviates potential cache thrashing due to false sharing.

Maintaining coarse-grain coherence states can reduce the implementation cost because less state information is recorded at the cache and memory side. It is worth pointing out that cache coherence protocols with coarse-grain states can be derived from protocols with fine-grain states such as Cachet. Both the soundness and liveness of the derived protocols are automatically guaranteed since all the coherence operations are legal mandatory or voluntary operations.

Chapter 8

Conclusions

This thesis has addressed several important issues regarding cache coherence protocols for shared memory multiprocessor systems. First, what memory model should be supported to allow efficient and flexible implementations; second, what adaptivity can be provided to accommodate programs with different access patterns; and third, how adaptive cache coherence protocols can be designed and verified. We have proposed a scalable mechanism-oriented memory model called Commit-Reconcile & Fences (CRF), and developed an adaptive cache coherence protocol called Cachet that implements CRF in DSM system. Our research uses Term Rewriting Systems (TRSs) as the formalism to precisely specify memory models and cache coherence protocols.

The CRF memory model exposes data replication via a notion of semantic caches, referred to as saches. There are three types of memory instructions: memory access instructions Loadl and Storel, memory rendezvous instructions Commit and Reconcile, and memory fence instructions. Semantically, each processor is associated with a sache that contains a set of cells. Each sache cell has a state, which can be either Clean or Dirty. The Clean state indicates that the data has not been modified since it was cached or last written back, while the Dirty state indicates that the data has been modified and has not been written back to the memory since then. At any time, a sache can purge a Clean copy from the sache, write a Dirty copy back to the memory, or retrieve a Clean copy from the memory for an uncached address. A Commit instruction cannot be completed if the address is cached in the Dirty state, and a Reconcile instruction cannot be completed if the address is cached in the Clean state.

There are good reasons to be skeptical of yet another memory model. Memory model definitions in modern microprocessor manuals are sometimes imprecise, ambiguous or even wrong. Not only computer architects, but also compiler writers and system programmers, would prefer memory models to be simpler and cleaner at the architecture level. The CRF model can eliminate some *modèle de l'année* disadvantages of existing memory models. Programs written under sequential consistency and various weaker memory models can be translated into efficient CRF programs, while most existing parallel systems can be interpreted as specific implementations of CRF, though more efficient implementations are possible. Indeed, the

CRF model has both upward and downward compatibility, that is, the ability to run existing programs correctly and efficiently on a CRF machine, and the ability to run CRF programs well on existing machines.

The CRF model was motivated from a directory-based cache coherence protocol that was originally designed for the MIT Start-Voyager multiprocessor system [11, 12]. The protocol verification requires a memory model that specifies the legal memory behaviors that the protocol was supposed to implement. Ironically, we did not have such a specification even after the protocol design was completed. As a result, it was not clear what invariants must be proved in order to ensure that the cache coherence protocol always exhibits the same behavior as sequential consistency for properly synchronized programs. The lack of a precise specification also made it difficult to understand and reason about the system behavior for programs that may have data races. To deal with this dilemma, we transformed the protocol by eliminating various directive operations and implementation details such as message queues, preserving the semantics throughout the transformation. This eventually led to CRF, an abstract protocol that cannot be further simplified. The CRF model can be implemented efficiently for shared memory systems, largely because the model itself is an abstraction of a highly optimized cache coherence protocol.

We have designed a set of adaptive cache coherence protocols which are optimized for some common access patterns. The Base protocol is the most straightforward implementation of CRF, and is ideal for programs in which only necessary commit and reconcile operations are performed. The WP protocol allows a reconcile operation on a clean cache cell to complete without purging the cell so that the data can be accessed by subsequent load operations without causing a cache miss; this is intended for programs that contain excessive reconcile operations. The Migratory protocol allows an address to be cached in at most one cache; it fits well when one processor is likely to access an address many times before another processor accesses the same address.

We further developed an adaptive cache coherence protocol called Cachet that provides a wide scope of adaptivity for DSM systems. The Cachet protocol is a seamless integration of multiple micro-protocols, and embodies both intra-protocol and inter-protocol adaptivity to achieve high performance under changing memory access patterns. A cache cell can be modified without the so-called exclusive ownership, which effectively allows multiple writers for the same memory location simultaneously. This can reduce the average latency for write operations and alleviate potential cache thrashing due to false sharing. Moreover, the purge of an invalidated cache cell can be deferred to the next reconcile point, which can help reduce cache thrashing due to read-write false sharing. An early version of Cachet with only the Base and WP micro-protocols can be found elsewhere [113]. Since Cachet implements the CRF model, it is automatically a protocol that implements the memory models whose programs can be translated into CRF programs.

Our view of adaptive protocols contains three layers: mandatory rules, voluntary rules and heuristic policies. Mandatory rules are weakly or strongly fair to ensure the liveness of the system, while voluntary rules have no such requirement and are used to specify adaptive actions without giving particular enabling conditions. Therefore, we can also think of Cachet as a family of cache coherence protocols in that a heuristic policy for selecting adaptive operations defines a complete protocol tailored for some particular access patterns. Different heuristic policies can exhibit different performance, but can never affect the soundness and liveness of the protocol.

We have proposed the two-stage Imperative-&-Directive methodology that separates the soundness and liveness concerns throughout protocol design and verification. The first stage involves only imperative rules that specify coherence actions that determine the soundness of the system. The second stage involves directive rules and the integration of imperative and directive rules to ensure both the soundness and liveness simultaneously. The Imperative-&-Directive methodology can dramatically simplify the design and verification of sophisticated cache coherence protocols, because only imperative rules need to be considered while verifying soundness properties. This novel methodology was first applied to the design of a cache coherence protocol for a DSM system with multi-level caches [110].

The following table illustrates the number of imperative and integrated rules for Cachet and its micro-protocols. As can be seen, although Cachet consists of about 150 rewriting rules, it has only 60 basic imperative rules that must be considered in the soundness proof, including many soundness-related invariants that are used in the liveness proof.

Protocol	Basic Rules		Composite Rules	
	Imperative Rules	Integrated Rules	Imperative Rules	Integrated Rules
Base	15	27	-	-
WP	19	45	-	-
Migratory	16	36	-	-
Cachet	60	113	15	33

We have shown that TRSs can be used to specify and verify computer architectures and distributed protocols. While TRSs have something in common with some other formal techniques, we have found that the use of TRSs is more intuitive in both architecture descriptions and correctness proofs. The descriptions of micro-architectures and asynchronous protocols using TRSs are more precise than what one may find in a modern textbook [55]. The "Architectures via TRSs" technique was first applied to the specification and verification of a simple micro-architecture that employs register renaming and permits out-of-order instruction execution [111].

TRSs can be used to prove the correctness of an implementation with respect to a specification. The proof is based on simulation with respect to a mapping function. The mapping functions can be defined based on the notion of drained terms, which can be obtained via forward or backward draining or a combination of both. The invariants often show up systematically and can be verified by case analysis and simple induction. The promise of TRSs for computer architecture is the development of a set of integrated design tools for modeling, specification, verification, simulation and synthesis of computer systems.

8.1 Future Work

CRF Microprocessors We have used CRF as the specification for cache coherence protocols. The CRF model can be implemented on most modern microprocessors via appropriate translation schemes. However, it remains an open question how CRF instructions can be effectively incorporated into modern microprocessors. For example, what is the proper granularity for commit, reconcile and fence instructions? What optimizations can be performed by the compiler to eliminate unnecessary synchronizations? Since ordinary load and store instructions are decomposed into finer-grain instructions, the instruction bandwidth needed to support a certain level of performance is likely to be high. This can have profound impact on micro-architectures such as instruction dispatch, cache state access and cache snoopy mechanism. Another interesting question is the implementation of CRF instructions on architectures with malleable caches such as column and curious caching [29].

Optimizations of Cachet The Cachet protocol can be extended in many aspects to incorporate more adaptivity. For example, in Cachet, an instruction is always stalled when the cache cell is in a transient state. This constraint can be relaxed under certain circumstances: a Loadl instruction can complete if the cache state is WbPending, and a Commit instruction can complete if the cache state is CachePending.

The Cachet protocol uses a general cache request that draws no distinction between different micro-protocols. Although a cache can indicate what copy it prefers as heuristic information, the memory decides what copy to supply to the cache. We can extend the protocol so that in addition to the general cache request, a cache can also send a specific cache request for a specific type of cache copy. This can be useful when caches have more knowledge than the memory about the access patterns of the program. Another advantage of having distinct cache requests is that a cache can send a request for a WP or Migratory copy while the address is cached in some Base state. In this case, the cache request behaves as an upgrade request from Base to WP or Migratory.

It is worth noting that Cachet does not allow a cache to request an upgrade operation from WP to Migratory; instead the cache must first downgrade the cell from WP to Base and then send a cache request to the memory (although the downgrade message can be piggybacked with the cache request). We can introduce an upgrade request message so that a cache can upgrade a WP cell to a Migratory cell without first performing the downgrade operation (so that the memory does not need to send the data copy to the cache).

In Cachet, a cache can only receive a data copy from the memory, even though the most up-to-date data resides in another cache at the time. Therefore, a Migratory copy must be written back to the memory first before the data can be supplied to another cache. The forwarding technique can be used to allow a cache to retrieve a data copy directly from another cache. This can reduce the latency to service cache misses for programs that exhibit access patterns such as the producer-consumer pattern.

The Cachet protocol is designed for NUMA systems. It can be extended with COMA-like coherence operations to provide more adaptivity. This allows a cache to switch between NUMA and COMA styles for the same memory region dynamically.

Heuristic Policies The Cachet protocol provides enormous adaptivity for programs with various access patterns. A relevant question is what mechanisms and heuristic policies are needed to discover the access patterns and how appropriate heuristic information can be conveyed to protocol engines. Access patterns can be detected through compiler analysis or runtime statistic collection. The Cachet protocol defines a framework in which various heuristic policies can be examined while the correctness of the protocol is always guaranteed. Customized protocols can be built dynamically with guaranteed soundness and liveness.

Access patterns can also be given by the programmer as program annotations. The voluntary rules of Cachet represent a set of coherence primitives that can be safely invoked by programmers whenever necessary. Programmers can therefore build application specific protocols by selecting appropriate coherence primitives. The primitive selection is just a performance issue, and the correctness of the system can never be compromised, regardless of when and how the primitives are executed.

Automatic Verification and Synthesis of Protocols When a system or protocol has many rewriting rules, the correctness proofs can quickly become tedious and error-prone. This problem can be alleviated by the use of theorem provers and model checkers. We are currently using theorem provers such as PVS [95, 108] in our verification effort of sophisticated protocols such as the complete Cachet protocol. Theorem provers are usually better at proving things correct than at finding and diagnosing errors. Therefore, it can also be useful to be able to do initial "sanity checking" using finite-state verifiers such as Murphi [35] or SPIN [59]. This often requires scaling down the example so that it has a small number of finite-state processes.

TRS descriptions, augmented with proper information about the system building blocks, hold the promise of high-level synthesis. A TRS compiler [58] compiles high-level behavioral descriptions in TRSs into Verilog that can be simulated and synthesized using commercial tools. This can effectively reduce the hardware design hurdle by allowing direct synthesis of TRS descriptions. We are currently exploring hardware synthesis of cache coherence protocols based on their TRS specifications.

Appendix A

The Cachet Protocol Specification

Mandatory Processor Rules				
Instruction	**Cstate**	**Action**	**Next Cstate**	
Loadl(a)	Cell(a,v,Clean$_\mathrm{b}$)	*retire*	Cell(a,v,Clean$_\mathrm{b}$)	SF
	Cell(a,v,Dirty$_\mathrm{b}$)	*retire*	Cell(a,v,Dirty$_\mathrm{b}$)	SF
	Cell(a,v,Clean$_\mathrm{w}$)	*retire*	Cell(a,v,Clean$_\mathrm{w}$)	SF
	Cell(a,v,Dirty$_\mathrm{w}$)	*retire*	Cell(a,v,Dirty$_\mathrm{w}$)	SF
	Cell(a,v,Clean$_\mathrm{m}$)	*retire*	Cell(a,v,Clean$_\mathrm{m}$)	SF
	Cell(a,v,Dirty$_\mathrm{m}$)	*retire*	Cell(a,v,Dirty$_\mathrm{m}$)	SF
	Cell(a,v,WbPending)	*stall*	Cell(a,v,WbPending)	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	
	$a \notin$ *cache*	*stall*, $\langle$CacheReq,$a\rangle \rightarrow$ H	Cell(a,-,CachePending)	
Storel(a,v)	Cell(a,-,Clean$_\mathrm{b}$)	*retire*	Cell(a,v,Dirty$_\mathrm{b}$)	SF
	Cell(a,-,Dirty$_\mathrm{b}$)	*retire*	Cell(a,v,Dirty$_\mathrm{b}$)	SF
	Cell(a,-,Clean$_\mathrm{w}$)	*retire*	Cell(a,v,Dirty$_\mathrm{w}$)	SF
	Cell(a,-,Dirty$_\mathrm{w}$)	*retire*	Cell(a,v,Dirty$_\mathrm{w}$)	SF
	Cell(a,-,Clean$_\mathrm{m}$)	*retire*	Cell(a,v,Dirty$_\mathrm{m}$)	SF
	Cell(a,-,Dirty$_\mathrm{m}$)	*retire*	Cell(a,v,Dirty$_\mathrm{m}$)	SF
	Cell(a,v_1,WbPending)	*stall*	Cell(a,v_1,WbPending)	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	
	$a \notin$ *cache*	*stall*, $\langle$CacheReq,$a\rangle \rightarrow$ H	Cell(a,-,CachePending)	
Commit(a)	Cell(a,v,Clean$_\mathrm{b}$)	*retire*	Cell(a,v,Clean$_\mathrm{b}$)	SF
	Cell(a,v,Dirty$_\mathrm{b}$)	*stall*, $\langle$Wb$_\mathrm{b}$,a,$v\rangle \rightarrow$ H	Cell(a,v,WbPending)	
	Cell(a,v,Clean$_\mathrm{w}$)	*retire*	Cell(a,v,Clean$_\mathrm{w}$)	SF
	Cell(a,v,Dirty$_\mathrm{w}$)	*stall*, $\langle$Wb$_\mathrm{w}$,a,$v\rangle \rightarrow$ H	Cell(a,v,WbPending)	
	Cell(a,v,Clean$_\mathrm{m}$)	*retire*	Cell(a,v,Clean$_\mathrm{m}$)	SF
	Cell(a,v,Dirty$_\mathrm{m}$)	*retire*	Cell(a,v,Dirty$_\mathrm{m}$)	SF
	Cell(a,v,WbPending)	*stall*	Cell(a,v,WbPending)	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	
	$a \notin$ *cache*	*retire*	$a \notin$ *cache*	SF
Reconcile(a)	Cell(a,-,Clean$_\mathrm{b}$)	*retire*	$a \notin$ *cache*	SF
	Cell(a,v,Dirty$_\mathrm{b}$)	*retire*	Cell(a,v,Dirty$_\mathrm{b}$)	SF
	Cell(a,v,Clean$_\mathrm{w}$)	*retire*	Cell(a,v,Clean$_\mathrm{w}$)	SF
	Cell(a,v,Dirty$_\mathrm{w}$)	*retire*	Cell(a,v,Dirty$_\mathrm{w}$)	SF
	Cell(a,v,Clean$_\mathrm{m}$)	*retire*	Cell(a,v,Clean$_\mathrm{m}$)	SF
	Cell(a,v,Dirty$_\mathrm{m}$)	*retire*	Cell(a,v,Dirty$_\mathrm{m}$)	SF
	Cell(a,v,WbPending)	*stall*	Cell(a,v,WbPending)	
	Cell(a,-,CachePending)	*stall*	Cell(a,-,CachePending)	
	$a \notin$ *cache*	*retire*	$a \notin$ *cache*	SF

Figure A.1: Cachet: The Processor Rules

Voluntary C-engine Rules			
	Cstate	**Action**	**Next Cstate**
	Cell(a,-,Clean$_\mathrm{b}$)		$a \notin cache$
	Cell(a,v,Dirty$_\mathrm{b}$)	$\langle$Wb$_\mathrm{b}$,a,$v\rangle \to$ H	Cell(a,v,WbPending)
	Cell(a,v,Clean$_\mathrm{w}$)	$\langle$Down$_\mathrm{wb}$,$a\rangle \to$ H	Cell(a,v,Clean$_\mathrm{b}$)
	Cell(a,v,Dirty$_\mathrm{w}$)	$\langle$Down$_\mathrm{wb}$,$a\rangle \to$ H	Cell(a,v,Dirty$_\mathrm{b}$)
		$\langle$Wb$_\mathrm{w}$,a,$v\rangle \to$ H	Cell(a,v,WbPending)
	Cell(a,v,Clean$_\mathrm{m}$)	$\langle$Down$_\mathrm{mw}$,$a\rangle \to$ H	Cell(a,v,Clean$_\mathrm{w}$)
		$\langle$Down$_\mathrm{mb}$,$a\rangle \to$ H	Cell(a,v,Clean$_\mathrm{b}$)
	Cell(a,v,Dirty$_\mathrm{m}$)	$\langle$DownV$_\mathrm{mw}$,a,$v\rangle \to$ H	Cell(a,v,Clean$_\mathrm{w}$)
		$\langle$DownV$_\mathrm{mb}$,a,$v\rangle \to$ H	Cell(a,v,Clean$_\mathrm{b}$)
	$a \notin cache$	$\langle$CacheReq,$a\rangle \to$ H	Cell(a,-,CachePending)

Mandatory C-engine Rules			
Msg from H	**Cstate**	**Action**	**Next Cstate**
$\langle$Cache$_\mathrm{b}$,a,$v\rangle$	Cell(a,-,CachePending)		Cell(a,v,Clean$_\mathrm{b}$)
$\langle$Cache$_\mathrm{w}$,a,$v\rangle$	Cell(a,-,Clean$_\mathrm{b}$)		Cell(a,v,Clean$_\mathrm{w}$)
	Cell(a,v_1,Dirty$_\mathrm{b}$)		Cell(a,v_1,Dirty$_\mathrm{w}$)
	Cell(a,v_1,WbPending)		Cell(a,v_1,WbPending)
	Cell(a,-,CachePending)		Cell(a,v,Clean$_\mathrm{w}$)
	$a \notin cache$		Cell(a,v,Clean$_\mathrm{w}$)
$\langle$Cache$_\mathrm{m}$,a,$v\rangle$	Cell(a,-,Clean$_\mathrm{b}$)		Cell(a,v,Clean$_\mathrm{m}$)
	Cell(a,v_1,Dirty$_\mathrm{b}$)		Cell(a,v_1,Dirty$_\mathrm{m}$)
	Cell(a,v_1,WbPending)		Cell(a,v_1,WbPending)
	Cell(a,-,CachePending)		Cell(a,v,Clean$_\mathrm{m}$)
	$a \notin cache$		Cell(a,v,Clean$_\mathrm{m}$)
$\langle$Up$_\mathrm{wm}$,$a\rangle$	Cell(a,v,Clean$_\mathrm{b}$)		Cell(a,v,Clean$_\mathrm{b}$)
	Cell(a,v,Dirty$_\mathrm{b}$)		Cell(a,v,Dirty$_\mathrm{b}$)
	Cell(a,v,Clean$_\mathrm{w}$)		Cell(a,v,Clean$_\mathrm{m}$)
	Cell(a,v,Dirty$_\mathrm{w}$)		Cell(a,v,Dirty$_\mathrm{m}$)
	Cell(a,v,WbPending)		Cell(a,v,WbPending)
	Cell(a,-,CachePending)		Cell(a,-,CachePending)
	$a \notin cache$		$a \notin cache$
$\langle$WbAck$_\mathrm{b}$,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean$_\mathrm{b}$)
$\langle$WbAck$_\mathrm{w}$,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean$_\mathrm{w}$)
$\langle$WbAck$_\mathrm{m}$,$a\rangle$	Cell(a,v,WbPending)		Cell(a,v,Clean$_\mathrm{m}$)
$\langle$DownReq$_\mathrm{wb}$,$a\rangle$	Cell(a,v,Clean$_\mathrm{b}$)		Cell(a,v,Clean$_\mathrm{b}$)
	Cell(a,v,Dirty$_\mathrm{b}$)		Cell(a,v,Dirty$_\mathrm{b}$)
	Cell(a,v,Clean$_\mathrm{w}$)	$\langle$Down$_\mathrm{wb}$,$a\rangle \to$ H	Cell(a,v,Clean$_\mathrm{b}$)
	Cell(a,v,Dirty$_\mathrm{w}$)	$\langle$Down$_\mathrm{wb}$,$a\rangle \to$ H	Cell(a,v,Dirty$_\mathrm{b}$)
	Cell(a,v,WbPending)		Cell(a,v,WbPending)
	Cell(a,-,CachePending)		Cell(a,-,CachePending)
	$a \notin cache$		$a \notin cache$
$\langle$DownReq$_\mathrm{mw}$,$a\rangle$	Cell(a,v,Clean$_\mathrm{b}$)		Cell(a,v,Clean$_\mathrm{b}$)
	Cell(a,v,Dirty$_\mathrm{b}$)		Cell(a,v,Dirty$_\mathrm{b}$)
	Cell(a,v,Clean$_\mathrm{w}$)		Cell(a,v,Clean$_\mathrm{w}$)
	Cell(a,v,Dirty$_\mathrm{w}$)		Cell(a,v,Dirty$_\mathrm{w}$)
	Cell(a,v,Clean$_\mathrm{m}$)	$\langle$Down$_\mathrm{mw}$,$a\rangle \to$ H	Cell(a,v,Clean$_\mathrm{w}$)
	Cell(a,v,Dirty$_\mathrm{m}$)	$\langle$DownV$_\mathrm{mw}$,a,$v\rangle \to$ H	Cell(a,v,Clean$_\mathrm{w}$)
	Cell(a,v,WbPending)		Cell(a,v,WbPending)
	Cell(a,-,CachePending)		Cell(a,-,CachePending)
	$a \notin cache$		$a \notin cache$
$\langle$DownReq$_\mathrm{mb}$,$a\rangle$	Cell(a,v,Clean$_\mathrm{b}$)		Cell(a,v,Clean$_\mathrm{b}$)
	Cell(a,v,Dirty$_\mathrm{b}$)		Cell(a,v,Dirty$_\mathrm{b}$)
	Cell(a,v,Clean$_\mathrm{w}$)	$\langle$Down$_\mathrm{wb}$,$a\rangle \to$ H	Cell(a,v,Clean$_\mathrm{b}$)
	Cell(a,v,Dirty$_\mathrm{w}$)	$\langle$Down$_\mathrm{wb}$,$a\rangle \to$ H	Cell(a,v,Dirty$_\mathrm{b}$)
	Cell(a,v,Clean$_\mathrm{m}$)	$\langle$Down$_\mathrm{mb}$,$a\rangle \to$ H	Cell(a,v,Clean$_\mathrm{b}$)
	Cell(a,v,Dirty$_\mathrm{m}$)	$\langle$DownV$_\mathrm{mb}$,a,$v\rangle \to$ H	Cell(a,v,Clean$_\mathrm{b}$)
	Cell(a,v,WbPending)		Cell(a,v,WbPending)
	Cell(a,-,CachePending)		Cell(a,-,CachePending)
	$a \notin cache$		$a \notin cache$

Figure A.2: Cachet: The Cache Engine Rules

Voluntary M-engine Rules			
	Mstate	**Action**	**Next Mstate**
	$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])\ (id \notin dir)$	$\langle \mathrm{Cache_w},a,v\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{C_w}[id \mid dir])$
	$\mathrm{Cell}(a,v,\mathrm{C_w}[id])$	$\langle \mathrm{Up_{wm}},a\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$
	$\mathrm{Cell}(a,v,\mathrm{C_w}[\epsilon])$	$\langle \mathrm{Cache_m},a,v\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$
	$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])$	$\langle \mathrm{DownReq_{wb}},a\rangle \rightarrow dir$	$\mathrm{Cell}(a,v,\mathrm{T_w}[dir,\epsilon])$
	$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$	$\langle \mathrm{DownReq_{mw}},a\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{T'_m}[id])$
		$\langle \mathrm{DownReq_{mb}},a\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{T_m}[id,\epsilon])$
	$\mathrm{Cell}(a,v,\mathrm{T'_m}[id])$	$\langle \mathrm{DownReq_{wb}},a\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{T_m}[id,\epsilon])$

Mandatory M-engine Rules				
Msg from id	**Mstate**	**Action**	**Next Mstate**	
$\langle \mathrm{CacheReq},a\rangle$	$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])\ (id \notin dir)$	$\langle \mathrm{Cache_b},a,v\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])$	SF
	$\mathrm{Cell}(a,v,\mathrm{T_w}[dir,sm])\ (id \notin dir)$	*stall message*	$\mathrm{Cell}(a,v,\mathrm{T_w}[dir,sm])$	
	$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])\ (id \in dir)$		$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])$	
	$\mathrm{Cell}(a,v,\mathrm{T_w}[dir,sm])\ (id \in dir)$		$\mathrm{Cell}(a,v,\mathrm{T_w}[dir,sm])$	
	$\mathrm{Cell}(a,v,\mathrm{C_m}[id_1])\ (id \neq id_1)$	*stall message* $\langle \mathrm{DownReq_{mw}},a\rangle \rightarrow id_1$	$\mathrm{Cell}(a,v,\mathrm{T'_m}[id_1])$	
	$\mathrm{Cell}(a,v,\mathrm{T'_m}[id_1])\ (id \neq id_1)$	*stall message*	$\mathrm{Cell}(a,v,\mathrm{T'_m}[id_1])$	
	$\mathrm{Cell}(a,v,\mathrm{T_m}[id_1,sm])\ (id \neq id_1)$	*stall message*	$\mathrm{Cell}(a,v,\mathrm{T_m}[id_1,sm])$	
	$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$	
	$\mathrm{Cell}(a,v,\mathrm{T'_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{T'_m}[id])$	
	$\mathrm{Cell}(a,v,\mathrm{T_m}[id,sm])$		$\mathrm{Cell}(a,v,\mathrm{T_m}[id,sm])$	
$\langle \mathrm{Wb_b},a,v\rangle$	$\mathrm{Cell}(a,v_1,\mathrm{C_w}[dir])\ (id \notin dir)$	$\langle \mathrm{DownReq_{wb}},a\rangle \rightarrow dir$	$\mathrm{Cell}(a,v_1,\mathrm{T_w}[dir,(id,v)])$	
	$\mathrm{Cell}(a,v_1,\mathrm{T_w}[dir,sm])\ (id \notin dir)$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[dir,(id,v) \mid sm])$	
	$\mathrm{Cell}(a,v_1,\mathrm{C_w}[id \mid dir])$	$\langle \mathrm{DownReq_{wb}},a\rangle \rightarrow dir$	$\mathrm{Cell}(a,v_1,\mathrm{T_w}[dir,(id,v)])$	
	$\mathrm{Cell}(a,v_1,\mathrm{T_w}[id \mid dir,sm])$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[dir,(id,v) \mid sm])$	
	$\mathrm{Cell}(a,v_1,\mathrm{C_m}[id_1])\ (id \neq id_1)$	$\langle \mathrm{DownReq_{mb}},a\rangle \rightarrow id_1$	$\mathrm{Cell}(a,v_1,\mathrm{T_m}[id_1,(id,v)])$	
	$\mathrm{Cell}(a,v_1,\mathrm{T'_m}[id_1])\ (id \neq id_1)$	$\langle \mathrm{Down_{wb}},a\rangle \rightarrow id_1$	$\mathrm{Cell}(a,v_1,\mathrm{T_m}[id_1,(id,v)])$	
	$\mathrm{Cell}(a,v_1,\mathrm{T_m}[id_1,sm])\ (id \neq id_1)$		$\mathrm{Cell}(a,v_1,\mathrm{T_m}[id_1,(id,v) \mid sm])$	
	$\mathrm{Cell}(a,v_1,\mathrm{C_m}[id])$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[\epsilon,(id,v)])$	
	$\mathrm{Cell}(a,v_1,\mathrm{T'_m}[id])$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[\epsilon,(id,v)])$	
	$\mathrm{Cell}(a,v_1,\mathrm{T_m}[id,sm])$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[\epsilon,(id,v) \mid sm])$	
$\langle \mathrm{Wb_w},a,v\rangle$	$\mathrm{Cell}(a,v_1,\mathrm{C_w}[id \mid dir])$	$\langle \mathrm{DownReq_{wb}},a\rangle \rightarrow dir$	$\mathrm{Cell}(a,v_1,\mathrm{T_w}[dir,(id,v)])$	
	$\mathrm{Cell}(a,v_1,\mathrm{T_w}[id \mid dir,sm])$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[dir,(id,v) \mid sm])$	
	$\mathrm{Cell}(a,v_1,\mathrm{C_m}[id])$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[\epsilon,(id,v)])$	
	$\mathrm{Cell}(a,v_1,\mathrm{T'_m}[id])$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[\epsilon,(id,v)])$	
	$\mathrm{Cell}(a,v_1,\mathrm{T_m}[id,sm])$		$\mathrm{Cell}(a,v_1,\mathrm{T_w}[\epsilon,(id,v) \mid sm])$	
$\langle \mathrm{Down_{wb}},a\rangle$	$\mathrm{Cell}(a,v,\mathrm{C_w}[id \mid dir])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[dir])$	
	$\mathrm{Cell}(a,v,\mathrm{T_w}[id \mid dir,sm])$		$\mathrm{Cell}(a,v,\mathrm{T_w}[dir,sm])$	
	$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[\epsilon])$	
	$\mathrm{Cell}(a,v,\mathrm{T'_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[\epsilon])$	
	$\mathrm{Cell}(a,v,\mathrm{T_m}[id,sm])$		$\mathrm{Cell}(a,v,\mathrm{T_w}[\epsilon,sm])$	
$\langle \mathrm{Down_{mw}},a\rangle$	$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[id])$	
	$\mathrm{Cell}(a,v,\mathrm{T'_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[id])$	
	$\mathrm{Cell}(a,v,\mathrm{T_m}[id,sm])$		$\mathrm{Cell}(a,v,\mathrm{T_w}[id,sm])$	
$\langle \mathrm{DownV_{mw}},a,v\rangle$	$\mathrm{Cell}(a,\text{-},\mathrm{C_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[id])$	
	$\mathrm{Cell}(a,\text{-},\mathrm{T'_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[id])$	
	$\mathrm{Cell}(a,\text{-},\mathrm{T_m}[id,sm])$		$\mathrm{Cell}(a,v,\mathrm{T_w}[id,sm])$	
$\langle \mathrm{Down_{mb}},a\rangle$	$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[\epsilon])$	
	$\mathrm{Cell}(a,v,\mathrm{T'_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[\epsilon])$	
	$\mathrm{Cell}(a,v,\mathrm{T_m}[id,sm])$		$\mathrm{Cell}(a,v,\mathrm{T_w}[\epsilon,sm])$	
$\langle \mathrm{DownV_{mb}},a,v\rangle$	$\mathrm{Cell}(a,\text{-},\mathrm{C_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[\epsilon])$	
	$\mathrm{Cell}(a,\text{-},\mathrm{T'_m}[id])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[\epsilon])$	
	$\mathrm{Cell}(a,\text{-},\mathrm{T_m}[id,sm])$		$\mathrm{Cell}(a,v,\mathrm{T_w}[\epsilon,sm])$	
	$\mathrm{Cell}(a,\text{-},\mathrm{T_w}[\epsilon,(id,v) \mid sm])$	$\langle \mathrm{WbAck_b},a\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{T_w}[\epsilon,sm])$	
	$\mathrm{Cell}(a,\text{-},\mathrm{T_w}[\epsilon,(id,v)])$	$\langle \mathrm{WbAck_w},a\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{C_w}[id])$	
	$\mathrm{Cell}(a,\text{-},\mathrm{T_w}[\epsilon,(id,v)])$	$\langle \mathrm{WbAck_m},a\rangle \rightarrow id$	$\mathrm{Cell}(a,v,\mathrm{C_m}[id])$	
	$\mathrm{Cell}(a,v,\mathrm{T_w}[\epsilon,\epsilon])$		$\mathrm{Cell}(a,v,\mathrm{C_w}[\epsilon])$	

Figure A.3: Cachet: The Memory Engine Rules

FIFO Message Passing

$$msg_1 \otimes msg_2 \quad \equiv \quad msg_2 \otimes msg_1$$
$$\textit{if} \quad \mathrm{Dest}(msg_1) \neq \mathrm{Dest}(msg_2) \ \vee \ \mathrm{Addr}(msg_1) \neq \mathrm{Addr}(msg_2)$$

Buffer Management

$$msg_1 \odot msg_2 \quad \equiv \quad msg_2 \odot msg_1$$
$$\begin{aligned} \textit{if} \quad & (\mathrm{Cmd}(msg_1) = \mathrm{CacheReq} \ \vee \ \mathrm{Cmd}(msg_2) = \mathrm{CacheReq}) \ \wedge \\ & (\mathrm{Src}(msg_1) \neq \mathrm{Src}(msg_2) \ \vee \ \mathrm{Addr}(msg_1) \neq \mathrm{Addr}(msg_2)) \end{aligned}$$

Figure A.4: FIFO Message Passing and Buffer Management

Bibliography

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, pages 66–76, Dec. 1996.

[2] S. V. Adve and M. D. Hill. Weak Ordering – A New Definition. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 2–14, June 1990.

[3] S. V. Adve and M. D. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, June 1993.

[4] Y. Afek, G. Brown, and M. Merritt. Lazy Caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, Jan. 1993.

[5] A. Agarwal, R. Bianchini, D. Chaiken, kirk Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[6] A. Agarwal, R. Simon, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280–289, May 1988.

[7] H. Akhiani, D. Doligez, P. Harter, L. Lamport, J. Scheid, M. Tuttle, and Y. Yu. Cache Coherence Verification with TLA+. In *World Congress on Formal Methods in the Development of Computing Systems, Industrial Panel*, Toulouse, France, Sept. 1999.

[8] C. Amza, A. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive Protocols for Software Distributed Shared Memory. *Proceedings of IEEE, Special Issue on Distributed Shared Memory*, 87(3):467–475, Mar. 1999.

[9] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[10] B. S. Ang. Design and Implementation of a Multi-purpose Cluster System Network Interface Unit. PhD Dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Feb. 1999.

[11] B. S. Ang, D. Chiou, D. Rosenband, M. Ehrlich, L. Rudolph, and Arvind. StarT-Voyager: A Flexible Platform for Exploring Scalable SMP Issues. In *Supercomputing*, Nov. 1998.

[12] B. S. Ang, D. Chiou, L. Rudolph, and Arvind. The StarT-Voyager Parallel System. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, Oct. 1998.

[13] J. K. Archibald. The Cache Coherence Problem in Shared-Memory Multiprocessors. PhD Dissertation, Department of Computer Science, University of Washington, Feb. 1987.

[14] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[15] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.

[16] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE COMPCON*, 1993.

[17] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, June 1996.

[18] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-Consistent Distributed Shared Memory. In *Proceedings of the 10th International Parallel Processing Symposium*, Apr. 1996.

[19] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, July 1995.

[20] M. Blumrich, R. Alpert, Y. Chen, D. Clark, S. Damianakis, C. Dubnicki, E. Felten, L. Iftode, K. Li, M. Martonosi, and R. Shillner. Design Choices in the SHRIMP System: An Empirical Study. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[21] G. M. Brown. Asynchronous Multicaches. *Distributed Computing*, 4:31–36, 1990.

[22] M. Browne, E. Clarke, D. Dill, and B. Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Transaction on Computers*, pages 1035–1044, Dec. 1986.

[23] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *International Conference on Computer-Aided Verification*, June 1994.

[24] J. B. Carter, A. Davis, R. Kuramkote, C. C. Kuo, L. B. Stoller, and M. .Swanson. Avalanche: A Communication and Memory Architecture for Scalable Parallel Computing. In *Proceedings of the 5th Workshop on Scalable Shared Memory Multiprocessors*, June 1995.

[25] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, Dec. 1978.

[26] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based Cache Coherence in Large-scale Multiprocessors. *Computer*, pages 49–58, June 1990.

[27] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, Apr. 1991.

[28] S. Chandra, B. Richard, and J. R. Larus. Teapot: Language support for writing memory coherence protocols. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, May 1996.

[29] D. Chiou. Extending the Reach of Microprocessors: Column and Curious Caching. PhD Dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1999.

[30] E. Clarke, E. Emerson, and A. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, Apr. 1986.

[31] A. E. Condon, M. D. Hill, M. Plakal, and D. J. Sorin. Using Lamport Clocks to Reason About Relaxed Memory Models. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, 1999.

[32] B. Cook, J. Launchbury, and J. Matthews. Specifying Superscalar Microprocessors in Hawk. In *Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems, Marstrand, Sweden*, June 1998.

[33] F. Corella, J. M. Stone, and C. Barton. A Formal Specification of the PowerPC Shared Memory Architecture. Research Report 18638 (81566), IBM Research Division, 1993.

[34] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.

[35] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[36] D. L. Dill, S. Park, and A. G. Nowatzyk. Formal Specification of Abstract Memory Models. In *Research on Integrated Systems: Proceedings of the 1993 Symposium, MIT Press*, pages 38–52, 1993.

[37] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 434–442, June 1986.

[38] S. Eggers and R. H. Katz. Evaluating the Performance for Four Snooping Cache Coherency Protocols. In *Proceedings of the 16th International Symposium on Computer Architecture*, May 1989.

[39] S. J. Eggers and R. H. Katz. A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation. In *Proceedings of the 14th International Symposium on Computer Architecture*, May 1987.

[40] E. A. Emerson. Temporal and Modal Logic. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics.* MIT Press, 1990.

[41] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[42] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Supercomputing*, Nov. 1994.

[43] B. Falsafi and D. A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th International Symposium on Computer Architecture*, May 1997.

[44] G. R. Gao and V. Sarkar. Location Consistency – Stepping Beyond the Barriers of Memory Coherence and Serializability. Technical Memo 78, ACAPS Laboratory, School of Computer Science, McGill University, Dec. 1993.

[45] G. R. Gao and V. Sarkar. Location Consistency – A New Memory Model and Cache Coherence Protocol. Technical Memo 16, CAPSL Laboratory, Department of Electrical and Computer Engineering, University of Delaware, Feb. 1998.

[46] K. Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. PhD Dissertation, Stanford University, 1995.

[47] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Programming for Different Memory Consistency Models. In *Journal of Parallel and Distributed Computing*, pages 399–407, Aug. 1992.

[48] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, pages 355–364, Aug. 1991.

[49] K. Gharachorloo, A. Gupta, and J. Hennessy. Revision to "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors". Technical Report CSL-TR-93-568, Computer Systems Laboratory, Stanford University, Apr. 1993.

[50] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[51] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[52] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 9th International Symposium on Computer Architecture*, pages 124–131, June 1983.

[53] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 422–431, May 1988.

[54] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, Menlo Park, CA, 1996.

[55] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1996.

[56] High Performance Fortran Forum, editor. *High Performance Fortran Language Specification (Version 2.0).* 1997.

[57] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architecture Validation for Processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[58] J. C. Hoe and Arvind. Hardware Synthesis from Term Rewriting Systems. In *Proceedings of VLSI*, Lisbon, Portugal, Dec. 1999.

[59] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[60] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge Between Release Consistency and Entry Consistency. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[61] Intel Corporation. *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Manual.* 1996.

[62] Intel Corporation. *IA-64 Application Developer's Architecture Guide.* May 1999.

[63] C. Ip and D. Dill. Better Verification Through Symmetry. In *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and Their Applications*, pages 87–100, Apr. 1993.

[64] C. Ip and D. Dill. Efficient Verification of Symmetric Concurrent Systems. In *International Conference on Computer Design: VLSI in Computers and Processors*, Oct. 1993.

[65] T. Joe. COMA-F: A Non-Hierarchical Cache Only Memory Architecture. PhD Dissertation, Stanford University, Mar. 1995.

[66] D. R. Kaeli, N. K. Perugini, and J. M. Stone. Literature Survey of Memory Consistency Models. Research Report 18843 (82385), IBM Research Devision, 1993.

[67] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing A Cache Consistency Protocol. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 276–283, June 1985.

[68] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13–21, May 1992.

[69] J. W. Klop. Term Rewriting System. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[70] A. Krishnamurthy and K. Yelick. Optimizing Parallel SPMD Programs. In *Languages and Compilers for Parallel Computing*, 1994.

[71] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, Apr. 1994.

[72] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[73] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[74] L. Lamport. The Module Structure of TLA+. Technical Note 1996-002a, Compaq Systems Research Center, Sept. 1996.

[75] L. Lamport. The Operators of TLA+. Technical Note 1997-006a, Compaq Systems Research Center, June 1997.

[76] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[77] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[78] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Logic, Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan. 1993.

[79] D. E. Lenoski. The Design and Analysis of DASH: A Scalable Directory-based Multiprocessor. PhD Dissertation, Stanford University, Feb. 1992.

[80] J. Levitt and K. Olukotun. A Scalable Formal Verification Methodology for Pipelined Microprocessors. In *Proceedings of the 33rd ACM IEEE Design Automation Conference*, June 1996.

[81] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[82] M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th International Symposium on Microarchitecture*, Dec. 1996.

[83] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Load Value Prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[84] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[85] J.-W. Maessen, Arvind, and X. Shen. Improving the Java Memory Model Using CRF. CSG Memo 428, Laboratory for Computer Science, Massachusetts Institute of Technology, Nov. 1999.

[86] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for A New Family of RISC Processors.* Morgan Kaufmann, 1994.

[87] K. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. PhD Dissertation, Carnegie Mellon University, May 1992.

[88] K. L. McMillan. Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking. In *Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems, Marstrand, Sweden*, June 1998.

[89] S. S. Mukherjee and M. D. Hill. An Evaluation of Directory Protocols for Medium-Scale Shared-Memory Multiprocessors. In *Proceedings of the 8th ACM International Conference on Supercomputing*, July 1994.

[90] S. S. Mukherjee and M. D. Hill. Using Prediction to Accelerate Coherence Protocols. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[91] H. L. Muller, P. W. A. Stallard, and D. H. D. Warren. Implementing the Data Diffusion Machine Using Crossbar Routers. In *Proceedings of the 10th International Parallel Processing Symposium*, 1996.

[92] R. S. Nikhil and Arvind. *Programming in pH – A Parallel Dialect of Haskell.* MIT, 1998.

[93] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, W. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, Aug. 1995.

[94] B. W. O'Krafka and A. Richard. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 138–147, May 1990.

[95] S. Owre, N.Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert, editors. *PVS Language Reference.* SRI international, Sept. 1998.

[96] M. S. Papamarcos and J. H. Patel. A Low-Overhead Coherence Solution for Multiprocessors With Private Cache Memories. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 348–354, June 1984.

[97] S. Park and D. L. Dill. An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order). In *Proceedings of the 7th ACM Symposium on Parallel Algorithms and Architectures*, pages 34–41, July 1995.

[98] S. Park and D. L. Dill. Protocol Verification by Aggregation of Distributed Transactions. In *International Conference on Computer-Aided Verification*, July 1996.

[99] S. Park and D. L. Dill. Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[100] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[101] F. Pong and M. Dubois. A New Approach for the Verification of Cache Coherence Protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6, Aug. 1995.

[102] F. Pong and M. Dubois. Formal Verification of Delayed Consistency Protocols. In *Proceedings of the 10th International Parallel Processing Symposium*, Apr. 1996.

[103] F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois. Verifying Distributed Directory-based Cache Coherence Protocols: S3.mp, a Case Study. In *Proceedings of the European Conference on Parallel Computing*, 1995.

[104] J. B. Saxe, J. J. Horning, J. V. Guttag, and S. J. Garland. Using Transformations and Verification in Circuit Design. *Formal Methods in System Design*, 3(3), Dec. 1993.

[105] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[106] C. Scheurich and M. Dubois. Correct Memory Operation of Cache-based Multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 234–243, June 1987.

[107] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[108] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert, editors. *PVS Prover Guide*. SRI international, Sept. 1998.

[109] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, pages 282–312, Apr. 1988.

[110] X. Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. CSG Memo 398, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1997.

[111] X. Shen and Arvind. Modeling and Verification of ISA Implementations. In *Proceedings of the Australasian Computer Architecture Conference, Perth, Australia*, Feb. 1998.

[112] X. Shen and Arvind. Using Term Rewriting Systems to Design and Verify Processors. *IEEE Micro Special Issue on "Modeling and Validation of Microprocessors"*, May/June 1999.

[113] X. Shen, Arvind, and L. Rodolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. In *Proceedings of the 13th ACM International Conference on Supercomputing*, June 1999.

[114] R. Simoni and M. Horowitz. Modeling the Performance of Limited Pointers Directories for Cache Coherence. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 309–318, May 1991.

[115] R. T. Simoni. Cache Coherence Directories for Scalable Multiprocessors. PhD Dissertation, Stanford University, Mar. 1995.

[116] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44.

[117] R. L. Sites and R. T. Witek, editors. *Alpha AXP Architecture Reference Manual.* Butterworth-Heinemann, 1995.

[118] I. C. Society. *IEEE Standard for Scalable Coherent Interface.* 1993.

[119] P. Stenström, B. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.

[120] U. Stern and D. L. Dill. Automatic Verification of the SCI Cache Coherence Protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[121] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual (Version 9).* Prentice-Hall, 1994.

[122] W. D. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989.

[123] P. J. Windley. Formal Modeling and Verification of Microprocessors. *IEEE Transactions on Computers*, 44(1), Jan. 1995.

[124] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[125] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, pages 28–40, Apr. 1996.

[126] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23rd International Symposium on Computer Architecture*, 1996.