Rapid Designs for Cache Coherence Protocol Engines in Bluespec

by

Man Cheuk Ng

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

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

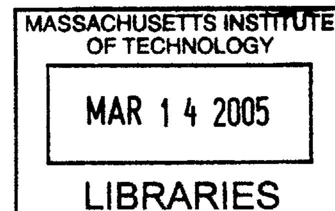
February 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
January 14, 2005

Certified by............................
...d
Johnson Professor
visor

Accepted by .. ..
Arthur C. Smith
Chairman, Department Committee on Graduate Students


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
MAR 14 2005
LIBRARIES
BARKER

Rapid Designs for Cache Coherence Protocol Engines in Bluespec

by

Man Cheuk Ng

Submitted to the Department of Electrical Engineering and Computer Science
on January 14, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

In this thesis, we present the framework for Rapid Protocol Engine Development (RaPED). We implemented the framework in Bluespec, which is a high level hardware language based on Term Rewriting Systems (TRSs). The framework is highly parameterized and general, thus allowing designers to design any protocol engine in a short period. Since protocol engines can be developed rapidly, designers can compare different designs instead of freezing the design prematurely in the development process.

We used the RaPED to implement a cache coherence protocol for Shen and Arvind's Commit-Reconcile and Fences (CRF) memory model [1]. The CRF allows scalable implementations of shared memory systems by decomposing memory access operations into simpler instructions. However, the focus for Shen's Cachet protocol for the CRF was adaptivity and correctness, it ignored some important implementation issues such as cache-line replacement, efficient buffer management and compatibility with multiword cache lines. In this thesis, we present a protocol called the Multiword Base protocol, which avoids these limitations. We defined the Multiword CRF (MCRF) memory model to help us to prove the correctness of Multiword Base. The MCRF is a specialization of the CRF with modifications that summarizes the properties of multiword cache lines. We show that Multiword Base is a correct implementation of the CRF by using the MCRF to simulate Multiword Base.

Apart from using multiword cache lines, many cache coherence protocols allow a cache to get data directly from another cache. The caches having this property is calling the snoopy caches. In this thesis, we present a CRF variant called the Snoopy CRF (SCRF) memory model, which gives hints to incorporate snoopy caches to the implementations of the CRF.

Thesis Supervisor: Arvind
Title: Johnson Professor

Acknowledgments

Funding for this work has been provided by the IBM agreement number W0133890 as a part of DARPA's PERCS Projects.

I extend my sincere gratitude and appreciation to many people who made this master's thesis possible. First of all, I would like to thank Arvind, my supervisor, for his guidance and advice on research. His creative thinking has inspired me and his endless enthusiasm on research has given me an example to follow.

I would like to thank all colleagues at the Computation Structures Group at MIT Computer Science and Artificial Intelligence Laboratory. I am highly indebted to my office-mate Byungsub Kim, who gave me inspiring research ideas and played baseball with me at late night in our research lab. Without him, I would not have withstood the feeling of loneliness at cold mid-night and continued to finish my work. Special thanks are due to Jacob Schwartz and Nirav Dave for their expertise in Bluspec. I would also like to thank the latter for answering my amateur Linux questions. Many thanks to John Sie Yuen Lee, Vinson Lee, Daihyun Lim, Jaewook Lee and Daniel L. Rosendband for making me feel home when I first came to MIT.

My deepest gratitude is owed to my girl friend Joanne for her continuing support even though we have been thousands of miles apart for a long time. Her patience and understanding helped me to keep focus on my research work through many difficult times.

Finally, I cannot thank my parents enough for giving me the opportunity to live in this wonderful world.

Contents

1 Introduction **13**

2 Framework for Rapid Protocol Engine Development (RaPED) **17**

2.1 Overview 17

2.2 Data Memory 18

2.2.1 DataReqMsg 19

2.2.2 DataResMsg 20

2.3 Network Controllers 21

2.3.1 Message Definitions of Network Controllers 21

2.3.2 Operational Semantics of Network Controllers 22

2.4 Protocol Processor 22

2.5 Implementation of RaPED in Bluespec 23

3 Multiword CRF **25**

3.1 MCRF Properties 26

3.2 MCRF Instructions and System Configurations 27

3.3 MCRF Rules 28

3.3.1 Terminology 30

3.4 Correctness Proof of the MCRF 34

4 Multiword Base Protocol **37**

4.1 Features 37

4.2 System Configurations 39

4.3 Definition of the Cache Site 41
4.3.1 Processor 41
4.3.2 Cache 42
4.3.3 Pend Queue 43
4.3.4 Stall Queue 45
4.3.5 Cache-side Incoming Message Buffer 46
4.3.6 Cache-side Outgoing Message Buffer 46
4.3.7 Cache-side Protocol Processor 46
4.4 Definition of the Memory Site 52
4.4.1 Memory 53
4.4.2 Memory-side Incoming Message Buffer 54
4.4.3 Memory-side Outgoing Message Buffer 54
4.4.4 Memory-side Protocol Processor 55
4.5 Definition of the Network 55
4.6 Correctness Proof of Multiword Base 56
4.6.1 Soundness Proof of Multiword Base 56
4.6.2 Liveness Proof of Multiword Base 59

5 Implementation of Multiword Base in RaPED 61

5.1 Processor Node 61
5.1.1 Network_{up} 63
5.1.2 Network_{low} 63
5.1.3 Data Memory 64
5.1.4 Protocol Processor 65
5.2 Memory Node 67
5.2.1 Data Memory 69
5.2.2 Network_{up} 69
5.2.3 Protocol Processor 69
5.3 Implementation of the Network 70

6 **Snoopy CRF** 71

6.1 SCRF Instructions and System Configurations 71

6.2 Rewrite Rules of the SCRF 72

6.3 Proof of the Correctness of the SCRF model 77

7 **Summary and Conclusions** 79

7.1 Future Work 84

A **Definitions of The CRF Model** 87

A.1 CRF Configurations 87

A.2 CRF Rules 87

A.2.1 CR Model 89

A.3 Rules for Instruction Reorderings and Memory Fences 90

List of Figures

2-1 Overview of RaPED 18

2-2 Definitions of DataReqMsg and DataResMsg 19

2-3 Operational Semantics of Data Memory 20

2-4 Definitions of Network Controllers Messages 22

2-5 Abstracted Operational Semantics of Protocol Processor 23

3-1 cache line Width of Commercial Processors 26

3-2 Correctness Issue of False Sharing 27

3-3 Instructions and System Configurations of the MCRF 29

3-4 Rewrite Rules Terminology 30

3-5 Summary of the MCRF Rules 34

3-6 Mapping MCRF rules to CRF rules 35

4-1 The System Configurations of Multiword Base 39

4-2 CRF Instructions 40

4-3 The Messages of Multiword Base 40

4-4 Instruction Reordering Table 42

4-5 Definitions of the Cache Operations 43

4-6 Operation Semantics of the Cache 44

4-7 Operations of pendQ 44

4-8 Operations of stallQ 45

4-9 CPP rules for Loadl and Storel 47

4-10 CPP rules for Commit and Reconcile 48

4-11 CPP Execution Sequences for Loadl 49

4-12 CPP Execution Sequences for Storel 50
4-13 CPP Execution Sequences for Commit 51
4-14 CPP Execution Sequences for Reconcile 52
4-15 Components of the Memory Site 53
4-16 Definitions of the Memory Operations 54
4-17 MPP Rules Summary 56
4-18 Limitation of Backward Draining (Convergence of Rules) 58
4-19 Limitation of Forward Draining (Divergence of Rules) 58
4-20 Mapping From Multiword Base to MCRF 59

5-1 Multiword Base in RaPED 62
5-2 Mapping Cache Site Components to the Modules of the Processor Node 62
5-3 Definitions of the Messages of the Processor Node 63
5-4 Pending Queue Implementation 66
5-5 Mapping Memory Site Components to the Modules of the Memory Node 68
5-6 Definitions of the Messages of the Memory Node 68

6-1 Instructions and System Configurations of the SCRF 73
6-2 Summary of the SCRF Rules 77
6-3 Mapping SCRF rules to CRF rules 78

7-1 Overview of RaPED 80
7-2 Summary of the MCRF Rules 81
7-3 Mapping MCRF rules to CRF rules 81
7-4 The Overview of the Implementation of Multiword Base 82
7-5 Summary of the SCRF Rules 83
7-6 Mapping SCRF rules to CRF rules 83

A-1 Instructions and System Configurations of CRF 88
A-2 Instruction Reordering Table 90

Chapter 1

Introduction

In 1965, Gordon Moore observed that the number of transistors per integrated circuit had doubled every couple of years. Although it was a prediction, this trend has lasted for nearly 40 years and is expected to hold true at least until the end of the decade. With the number of transistors per integrated circuit doubling every couple of years, computer architects have invented various techniques to use the extra transistors efficiently. These techniques include pipelining, instruction reordering, branch prediction, instruction speculation, value speculation, caching, and super-scalar execution. All the techniques increase the computation speed by making the Computer Processing Unit (CPU) execute more instructions simultaneously. However, these techniques have been explored thoroughly and do not have much room for improvement. For example, we cannot reorder too many instructions because reordering requires complicated and large amount of hardware to hold the state of executing instructions, which in turn may reduce performance.

A new technique called Chip Multi-Processor (CMP) has recently appeared in commodity high-performance computing. It improves system performance by implementing multiple processing cores in a single circuit chip. Each processing core has all the functionalities of a single CPU. Therefore, a computer with a multi-core processor is simply a shared memory multiprocessor system. IBM is among the pioneers to apply this technique in their designs. Its POWER4 and POWER5 have dual-core processors. Other corporations such as Intel and AMD will have their dual-

core designs for both desktop and server applications in 2005. In the past, shared memory multiprocessors systems required expensive networks to connect the processors and specialists for coding parallel programs. However, with the realization of CMP, shared memory multiprocessor systems are no longer unaffordable and rare. Therefore, efficient implementations of CMP are important for the future generations of computers.

One major factor for an efficient CMP implementation is cache coherence protocols. Caching reduces the average latency of memory operations by replicating frequently accessed data in storage units (caches) close to the processor. Caching is transparent to programmers in uniprocessor systems since all the optimizations of caching are designed not to affect the uniprocessor's memory model. However, the same optimizations are problematic to shared memory multiprocessor systems because they can produce different relaxed memory models for the multiprocessor systems. Therefore, cache coherent protocols are needed to ensure that each processor can observe the semantic effect of memory access operations performed by another processor in time.

We have developed the framework for Rapid Protocol Engine Development (RaPED) to help designers to implement cache coherence protocol rapidly and efficiently. RaPED distributes a cache coherence protocol engine into nodes and generalizes the implementation of the nodes. RaPED was implemented in the Bluespec language, which is a high level hardware language based on Term Rewriting Systems (TRSs). In 1999, James Hoe and Arvind showed that TRSs can be used to synthesis hardware circuit efficiently [2]. Bluespec was created based on this discovery. One advantage of Bluespec is that it is a language for both hardware simulation and synthesis: In traditional hardware design process, simulation and synthesis are written in different languages. Normally, simulations are written in high level software languages like C/C++ or Java; while hardware is done from RTL level hardware languages like Verilog or VHDL. Therefore, designers are required to write the code twice for a single design. Moreover, the semantic gap between the simulation language and the synthesis language makes it difficult for designers to prove that the two sources represent

the same design. Another advantage of Bluespec is that it accepts parameterized and modular designs, which enhance the re-usability of the code.

We have used RaPED to implement the Multi-word Base Protocol, which is a protocol for the Commit-Reconcile Fences (CRF) memory model [1]. The CRF exposes the notion of cache by decomposing memory operation into simpler instructions. In the CRF model, a memory load operation is decomposed into a Reconcile instruction followed by a Loadl instruction, and a memory store operation is decomposed into a Storel instruction followed by a Commit instruction. The decomposition allows the implementations of the CRF to be efficient and scalable. One reason is decomposed memory operations allow longer period for the system to carry out the coherence operations without affecting the semantics of the program. Another reason is the decomposition helps to reduce the number of coherence operations by eliminating unnecessary operations.

Multiword Base is different from other protocols for the CRF. It is the first CRF protocol that supports cache lines containing more than one address (multiword cache lines). Multiword Base is useful because multiword cache lines allows better usages of the cache. To prove the correctness of Multiword Base, we derived a variant of the CRF model: Multiword CRF (MCRF). The MCRF adds the properties of multiword cache lines to the CRF. By proving that Multiword Base can be simulated by the MCRF, we show that Multiword Base is a correct implementation of the CRF.

Snoopy cache is another common optimization for cache coherence protocols. It allows a cache to provide its data to another cache to reduce the cache miss penalty. We have derived another variant of the CRF model: Snoopy CRF (SCRF), which adds the properties of snoopy cache to the CRF. The SCRF gives us hints to incorporate snoopy caches to the implementation of the CRF.

This thesis is organized as follows: Following this introduction, Chapter 2 presents the definition of the RaPED. Chapter 3 defines the MCRF. Chapter 4 presents the Multiword Base protocol and presents the correctness proof for the protocol. Chapter 5 shows the implementation of Multiword Base in RaPED. Chapter 6 defines the SCRF. Finally, Chapter 7 presents the summary and conclusions.

Chapter 2

Framework for Rapid Protocol Engine Development (RaPED)

This chapter presents the framework for Rapid Protocol Engine Development (RaPED). Most protocol engine designs can be distributed into nodes. RaPED collects a bundle of interfaces to define the functionalities of a node. These interfaces are implemented in Bluespec. Since Bluespec allows parameterized and modular designs, RaPED provides a unique platform for designers to implement and evaluate their protocol engine designs efficiently. This chapter is organized as follows: In the Section 2.1, we give an overview of RaPED. In Section 2.2, Section 2.3 and Section 2.4, we describe each module in RaPED in details. In Section 2.5, I show how to convert the interfaces of RaPED into Bluespec implementation.

2.1 Overview

There are different designs for cache coherent multiprocessor systems. These designs can vary in different aspects. For example, different designs can have different coherence protocols, memory hierarchies and implementations of communication networks and caches. Therefore, it is difficult to define a single framework that generalizes all the cache coherence protocol engine designs. RaPED distributes a cache coherence protocol engine into nodes and generalizes the designs of the node. Figure 2-1

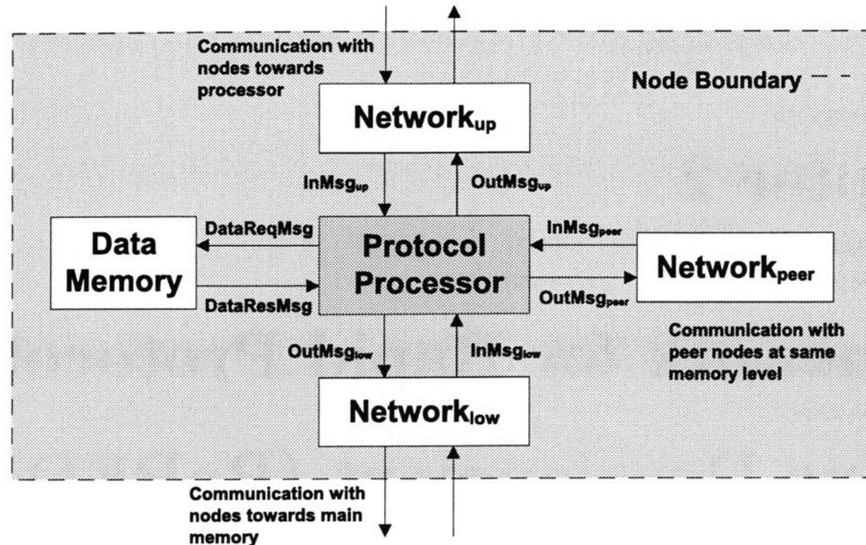


Figure 2-1: Overview of RaPED

shows the overview of RaPED. As can be seen, each node contains 5 modules: a data memory (**Data Memory**), three network controllers (**Network**$_{up}$, **Network**$_{peer}$ and **Network**$_{low}$) and a protocol processor (**Protocol Processor**). Modules shown in white are optional.

2.2 Data Memory

Data Memory stores the data and the book-keeping information (e.g. cache state) of memory addresses. It acts as the storage agent of **Protocol Processor**: **Protocol Processor** decides what data are stored in **Data Memory**, while **Data Memory** decides how data are stored. Therefore, **Protocol Processor** does not need to know every implementation details of **Data Memory**. For example, **Protocol Processor** probably does not need to know the cache associativity if **Data Memory** is a level 1 cache because cache associativity affects the performance instead of the functionality of the cache. Separating **Data Memory** from **Protocol Processor** also enhances

DataReqMsg	$\equiv$	Cache a v
	[]	Purge a
	[]	Read a
	[]	Update a v
DataResMsg	$\equiv$	CacheAck c a v
	[]	PurgeAck
	[]	ReadAck h v
	[]	UpdateAck h

Figure 2-2: Definitions of DataReqMsg and DataResMsg

the implementation flexibility because different cache designs can be used with the same Protocol Processor design. Moreover, this approach facilitates the verification process of **Protocol Processor** because the irrelevant implementation details of **Data Memory** are excluded.

There are four basic operations that **Data Memory** can perform: 1) Cache, 2) Purge, 3) Read and 4) Update. When **Protocol Processor** needs to access **Data Memory**, the former sends a Data Request Message (DataReqMsg) to the latter. Then, **Data Memory** answers the request with a Data Response Message (DataResMsg). Figure 2-2 shows the definitions of DataReqMsg and DataResMsg. The meanings of these messages are explained as follows.

2.2.1 DataReqMsg

There are four kinds of DataReqMsg. Each requests **Data Memory** to perform a type of operation:

1. **Cache** a v: cache the address a with data v.

2. **Purge** a: purge the address a.

3. **Read** a: report the data of the address a if the address is cached.

4. **Update** a v: overwrite the data of the address a to v if the address is cached.

Received Message	Current State	Reply Message	Next State
Cache a v	$(a,\text{-}) \notin$mem,no conflict	CacheAck False - -	$(a,v) \in$ mem
	$(a,\text{-}) \notin$mem, $(a',v')\in$mem, a and a' conflict	CacheAck True a' v'	$(a,v) \in$ mem, $(a',\text{-}) \notin$ mem
Purge a	$(a,\text{-}) \in$mem	PurgeAck	$(a,\text{-}) \notin$ mem
	$(a,\text{-}) \notin$mem	PurgeAck	$(a,\text{-}) \notin$ mem
Read a	$(a,v) \in$mem	ReadAck True v	$(a,v) \in$ mem
	$(a,\text{-}) \notin$mem	ReadAck False -	$(a,\text{-}) \notin$ mem
Update a v	$(a,\text{-}) \in$mem	WriteAck True	$(a,v) \in$ mem
	$(a,\text{-}) \notin$mem	WriteAck False	$(a,\text{-}) \notin$ mem

Figure 2-3: Operational Semantics of Data Memory

2.2.2 DataResMsg

There are also four kinds of DataResMsg, which response their corresponding DataReqMsg:

1. **CacheAck** c a v: report the completion of the cache operation. c is a boolean which is set if there is a cache replacement. If c is set, a is the replaced address and v is the data of the address.

2. **PurgeAck**: report the completion of the purge operation.

3. **ReadAck** h v: return the result of the Read operation. h is a boolean which is set if the address is stored in **Data Memory**. If h is set, v is the data of the address.

4. **UpdateAck** h: report the completion of the Update operation. h is a boolean which is set if the address is stored in **Data Memory**.

Clarifications for some special cases about the operations: 1) Purge has no effect if the address is uncached and 2) Update has no effect if the address is uncached. Figure 2-3 summarizes the operational semantics of **Data Memory**.

2.3 Network Controllers

There are three network controllers in each node, which are responsible for message passings between nodes:

1. **Network**$_{up}$: between the current node and the nodes at upper levels (towards the processor).

2. **Network**$_{peer}$: between the current node and the peer nodes at the same level.

3. **Network**$_{low}$: between the current node and the nodes at lower levels (towards the shared memory).

Similar to **Data Memory**, the three network controllers are separated from the main design to enhance the implementation flexibility. Most cache coherence protocol engine designs do not require **Protocol Processor** to know every implementation details of the network. These network controllers help **Protocol Processor** to send messages to their destinations. This approach simplifies the design and verification processes of **Protocol Processor** by avoiding the unnecessary complexity from the implementations of these network controllers. RaPED allows **Protocol Processor** to assume some properties for the network. For example, it can assume the network is fair or delivers messages in order. The implementations of the network controllers need to satisfy the assumptions made by **Protocol Processor** to guarantee the correctness.

2.3.1 Message Definitions of Network Controllers

Figure 2-4 defines the messages of the network controllers. When **Protocol Processor** is sending a message (OutMsg) to another node, it passes the message to the network controller. The message includes the identification of the destination *id* and the content *out*. On the other hand, when the network controller receives a message (InMsg) from another node, it forwards the message, which includes identification of the source *id* and the content *in*, to **Protocol Processor**.

OutMsg_{up}	$\equiv$	(id_{up}, out_{up})
InMsg_{up}	$\equiv$	(id_{up}, in_{up})
OutMsg_{peer}	$\equiv$	(id_{peer}, out_{peer})
InMsg_{peer}	$\equiv$	(id_{peer}, in_{peer})
OutMsg_{low}	$\equiv$	(id_{low}, out_{low})
InMsg_{low}	$\equiv$	(id_{low}, in_{low})

Figure 2-4: Definitions of Network Controllers Messages

2.3.2 Operational Semantics of Network Controllers

We can model a network controller with two queues that buffer the communication messages. One queue buffers the OutMsgs received from **Protocol Processor**. Another queue buffers the InMsgs received from the network. The main functionality of a network controller is buffer managements, which decide when to reorder messages. As mentioned, **Protocol Processor** can assume some properties to the message passing network. If the network cannot guarantee these properties, the network controllers can use buffer managements to assure the correctness. The following shows an example usage of the buffer management:

A cache coherence protocol requires message passings to be in order if the messages have the same destination and address. However, the message passing network used in the system cannot guarantee this property because the implementation makes it possible to reorder any message arbitrarily. To solve the problem, we can have a buffer management in the OutMsg queue which disallows messages having the same destination and address to enter the network simultaneously.

2.4 Protocol Processor

Protocol Processor connects with all other modules in the node. It gathers information from these modules and executes the corresponding coherence actions according to the protocol specifications. The operational semantics of the **Protocol Processor** are different for different protocols. However, **Protocol Processor** can be implemented as a lookup table. Figure 2-5 summarizes the inputs and outputs of

Lookup Table	
Input:	oldState, $\{\epsilon\|\text{InMsg}_{up}\}$, $\{\epsilon\|\text{InMsg}_{peer}\}$, $\{\epsilon\|\text{InMsg}_{low}\}$, $\{\epsilon\|\text{DataResMsg}\}$
Output:	newState, $\{\epsilon\|\text{OutMsg}_{up}\}$, $\{\epsilon\|\text{OutMsg}_{peer}\}$, $\{\epsilon\|\text{OutMsg}_{low}\}$, $\{\epsilon\|\text{DataReqMsg}\}$

Figure 2-5: Abstracted Operational Semantics of Protocol Processor

the lookup table. The inputs include the internal state of the protocol processor as well as the messages received from other modules. The outputs include the next state of the protocol processor and the messages to be sent to the connecting modules.

2.5 Implementation of RaPED in Bluespec

It is trivial to convert the definitions of RaPED into Bluespec definitions. For each module, there are two types of definitions: 1) Interface Definitions and 2) Message Definitions. The former defines all the inputs and outputs of a module and the latter defines the contents of these inputs and outputs. The following shows the conversion of the message and interface definitions to Bluespec code using **Data Memory** as an example:

Defining a Message in Bluespec

The following Bluespec expressions define the DataReqMsg in Figure 2-2:

Bluespec Code:

```
data DataReqMsg addr val =  Cache addr val |
                            Purge addr |
                            Read addr |
                            Update addr val
                            deriving (Bits, Eq)
```

The first line of the code defines a new data type called DataReqMsg, which requires two type variables *addr* and *val* to be set to actual types when it is used.

For an example, we can set both of them to "Bit 32" if the target system uses 32-bit addresses and 32-bit values. The next four lines defined the four possible values of DataReqMsg. The final line makes DataReqMsg expressible by bits and comparable. Other type of messages of RaPED can be defined similarly.

Defining an Interface in Bluespec

The following Bluespec expressions define the Data Memory interface:

Bluespec Code:

```
interface DataMemory addr val =
    putDataReq :: (DataReqMsg addr val) → Action
    getDataRes :: ActionValue (DataResMsg addr val)
```

The code creates a new interface called DataMemory. Similar to the definition of DataReqMsg, this interface requires two variables *addr* and *val* to be set to actual types when it is used. Two methods are defined for the interface: putDataReq and getDataRes. The former takes an input of type "DataReqMsg *addr val*" and then performs an action, which affects the internal state after execution. Meanwhile, the latter outputs a message of type "DataResMsg *addr val*" and then performs an action.

Chapter 3

Multiword CRF

In modern processor designs, a single cache line normally consists of multiple system values. Figure 3 shows the cache line width of several commodity processors. As can be seen, the cache lines are multiword wide, ranging from 64 bytes to 512 bytes. Multiword cache lines can reduce the overhead of the cache tags, which means more useful data can be stored in the cache. In general, programs have adequate spatial locality to ensure that most data in a cache line are accessed by the processor.

Since multiword cache lines are popular in processor designs, it is very useful to show that this optimization can be applied to the implementations of the CRF. Therefore, we have derived a variant of the CRF model called the Multiword CRF (MCRF). The MCRF is a specialization of the CRF which supports multiword cache lines. All possible execution behaviors of the MCRF model can be simulated by the CRF model. Therefore, a correct implementation of the MCRF automatically converts to a correct implementation of the CRF. The MCRF will be used in Chapter 4 to prove the correctness of an CRF protocol: the Multiword Base Protocol.

An advantage of the MCRF and the CRF models comparing to other memory is the possibility of avoiding the communication overhead of false sharings because the two models explicitly separate the data synchronizations from other memory operations. This allows the system to maintain copies of a cache line at different sites at the same time even when they are modifying the data of different addresses of the cache line.

After presenting the MCRF, we prove that the MCRF can be simulated by the original CRF.

Architecture	cache line Width
Intel Pentium 4 [8]	L1: 64 bytes, L2: 128 bytes
IBM Power4 [9]	L1, L2: 128 bytes, L3: 512 bytes

Figure 3-1: cache line Width of Commercial Processors

3.1 MCRF Properties

The following two properties make the MCRF support multiword cache line while maintaining the CRF semantics:

1. Each semantic cache line (sacheline) consists of the data of multiple consecutive addresses. At any time, a site either has all the addresses belonging to the same sacheline or has none of them.

2. Each sacheline maintains a cache state (CSTATE) for each address it contains.

Figure 3-2 gives an example showing the necessity of Property 2 to maintain correctness in the presence of false sharings. In the example, a program consisting of four CRF instructions is executed on two different CRF systems. "P1 Storel (a,v)" means that Storel is executed by processor 1 at address a with value v. "P2 Commit (a)" requires the address a to be committed by processor 2. The two systems have identical configurations, except that each sacheline in system 1 contains the data of one address while each sacheline in system 2 contains the data of two addresses. Moreover, a sacheline of either system maintains only one dirty bit, which is set if any address in the sacheline is updated (Storel). If there is a miss on Storel, data needed to be brought from the memory to the sache before the system can proceed. On the other hand, a dirty sacheline needs to be written back to the memory before Commit instruction can be retired. For more details about the CRF definitions , please refer to Appendix A. We can see that the two systems produce different results. System

2 loses the update from processor 1. This happens because the dirty bit assumes all addresses in the sacheline are overwritten by the processor 2. Therefore, Property 2 is necessary for the system to know the exact modified addresses.

Apart from the correctness issue, Property 2 also helps to avoid the communication overhead of false sharing. It is because this property allows the system to know whether different sites are accessing different addresses of the same sache line.

	System 1		System 2	
1: P1 Storel(0,5)	Sache 1:	addr0 (dirty, 5) addr1 (invalid,-)	Sache 1:	addr0&1 (dirty, (5, 0))
	Sache2:	addr0 (invalid,-) addr1 (invalid,-)	Sache2:	addr0&1 (invalid,-)
	Mem:	addr0 (0) addr1 (0)	Mem:	addr0 (0) addr1 (0)
2: P2 Storel(1,6)	Sache 1:	addr0 (dirty, 5) addr1 (invalid,-)	Sache 1:	addr0&1 (dirty, (5, 0))
	Sache2:	addr0 (invalid,-) addr1 (dirty, 6)	Sache2:	addr0&1 (dirty, (0, 6))
	Mem:	addr0 (0) addr1 (0)	Mem:	addr0 (0) addr1 (0)
3: P1 Commit(0)	Sache 1:	addr0 (clean, 5) addr1 (invalid,-)	Sache 1:	addr0&1 (clean, (5, 0))
	Sache2:	addr0 (invalid,-) addr1 (dirty, 6)	Sache2:	addr0&1 (dirty, (0, 6))
	Mem:	addr0 (5) addr1 (0)	Mem:	addr0 (5) addr1 (0)
4: P2 Commit(1)	Sache 1:	addr0 (clean, 5) addr1 (invalid,-)	Sache 1:	addr0&1 (clean, (5, 0))
	Sache2:	addr0 (invalid,-) addr1 (clean, 6)	Sache2:	addr0&1 (clean, (0, 6))
	Mem:	addr0 (5) addr1 (6)	Mem:	addr0 (0) addr1 (6)

Figure 3-2: Correctness Issue of False Sharing

3.2 MCRF Instructions and System Configurations

Figure 3-3 presents the instructions and system configurations of the MCRF. The instructions for the MCRF and the CRF are the same. There are eight instruc-

tions: Loadl, Storel, Reconcile, Commit, Fence_{rr}, Fence_{rw}, Fence_{wr}, Fence_{ww}. In the MCRF, a system contains a shared memory and a list of sites. Each site is composed of a processor (proc), a processor-to-memory buffer (pmb), a memory-to-processor buffer (mpb) and a semantic cache. The proc is responsible for sending the MCRF instructions to the pmb. The pmb buffers the messages delivered from the proc to the sache. Messages in pmb can be reordered unless there are data dependences or memory fences. On the other hand, the mpb buffers the results delivered from the sache to the proc. In contrast to pmb, messages in mpb can always be reordered arbitrarily. Each site is connected to the shared memory where the memory is used as the data rendezvous of the system. In the MCRF, the definitions of SITE and CELL are different from those in the CRF. In the MCRF, the definition of SITE has an extra parameter which specifies the width of the sachelines (CELL) in that site. For example, if this parameter is set to 4, each sacheline in that site will contain the data of four consecutive addresses which the first address is used to identify the sacheline. The MCRF allows different sites to have different sacheline widths, but sachelines within the same site must have the same width. The definition of CELL ensures that addresses belonging to the same sacheline are cached together, which is required by the MCRF Property 1. Moreover, each sacheline maintains a CSTATE for each address in the sacheline, which satisfies the MCRF Property 2.

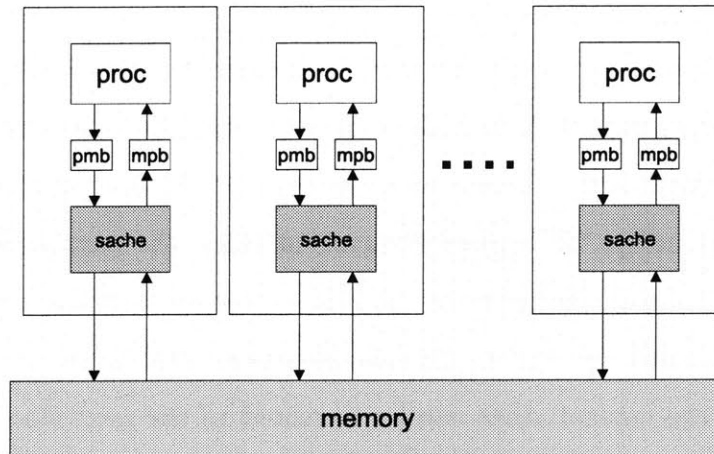
3.3 MCRF Rules

Same as the CRF, the MCRF has 2 sets of rules: The first set defines the operational semantics of Loadl, Storel, Commit and Reconcile instructions and some background rules that govern data propagations between semantic caches and memory. Meanwhile, the second set defines the semantics of instruction reorderings and memory fences. The MCRF rules only differ from the CRF rules by the first set. The two models have the same definitions for the second set. Therefore, we only present the definitions of the first set of rules in this thesis. We also discuss how each rule in the first set can be simulated by the original CRF rules. For reference, we have included

MCRF Instructions

INST	$\equiv$	Loadl(a) $[]$ Storel(a,v)
	$[]$	Commit(a) $[]$ Reconcile(a)
	$[]$	$\text{Fence}_{rr}(a1, a2)$ $[]$ $\text{Fence}_{rw}(a1, a2)$
	$[]$	$\text{Fence}_{wr}(a1, a2)$ $[]$ $\text{Fence}_{ww}(a1, a2)$

MCRF System Configurations



SYS	$\equiv$	Sys(MEM, SITEs)	*System*
SITEs	$\equiv$	SITE $[]$ SITE | SITEs	*Set of Sites*
SITE	$\equiv$	Site(n, SACHE, PMB, MPB, PROC)	*Site with cache of block size n*
SACHE	$\equiv$	ϵ $[]$ CELL| SACHE	*Semantic Cache*
CELL	$\equiv$	Cell(a,v_0,CSTATE,v_1,CSTATE,..,v_{n-1},CSTATE)	*cell of block size n*
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 3-3: Instructions and System Configurations of the MCRF

Term	Definition
sys if *cond* $\rightarrow$ *sys'*	the next configuration of *sys* can be *sys'* if the conditions specified by *cond* is satisfied
a div *b*	returns the integral of *a* divided by *h*
a mod *b*	returns the remainder of *a* divided by *h*
$\vee$	logical OR
$\wedge$	logical AND

Figure 3-4: Rewrite Rules Terminology

the definition of the original CRF model in Appendix A.

3.3.1 Terminology

Prior to the discussion of the MCRF rewrite rules, we show how a rule is defined. Table 3-4 summarizes the terms used to describe the rewrite rules and their definitions. The first row describes the format of a rewrite rule. In each rewrite rule, there are 3 parts: the part before "if" defines the current state; the part between "if" and the right arrow defines the condition for the rule execution; and the part after the right arrow defines the next state after the rule execution. The meaning of the rule can be interpreted as "the current state can be transited to the next state if the execution condition is satisfied". If several rewrite rules are applicable, the system will execute one of the rules arbitrarily. The second and third rows describe two arithmetic functions which return the integral and remainder of arithmetic division respectively. $\vee$ and $\wedge$ are "logical or" and "logical and" respectively.

Loadl and Storel Rules: A Loadl or Storel can be performed if the block containing the address is cached in the sache. A Loadl returns the data of the address to the processor through memory-to-processor buffer (mpb). A Storel instruction updates the data of the address in the sache and then acknowledges the processor through mpb.

MCRF-Loadl Rule

$$\begin{array}{llll} & \text{Site}(n, \text{sache}, \langle t, \text{Loadl}(a)\rangle; \text{pmb}, \text{mpb}, \text{proc}) & \text{if} & \text{Cell}(a', v_0, \text{-}, v_1, \text{-},.., v_m, \text{-},..) \in \text{sache} \\ & & \wedge & (a \text{ div } n) \equiv (a' \text{ div } n) \\ & & \wedge & m \equiv (a \text{ mod } n) \\ \rightarrow & \text{Site}(n, \text{sache}, \text{pmb}, \text{mpb}|\langle t, v_m\rangle, \text{proc}) & & \end{array}$$

MCRF-Storel Rule

$$\begin{array}{ll} & \text{Site}(n, \text{Cell}(a', v_0, c_0, v_1, c_1,.., v_m,\text{-},..)|\, \text{sache}, \langle t, \text{Storel}(a, v'_m)\rangle; \text{pmb}, \text{mpb}, \text{proc}) \\ & \text{if} \quad (a \text{ div } n) \equiv (a' \text{ div } n) \wedge m \equiv (a \text{ mod } n) \\ \rightarrow & \text{Site}(n, \text{Cell}(a', v_0, c_0, v_1, c_1,.., v'_m, \text{Dirty},..)|\, \text{sache}, \text{pmb}, \text{mpb}|\langle t, Ack\rangle, \text{proc}) \end{array}$$

Both MCRF-Loadl and MCRF-Storel can be simulated by CRF-Loadl and CRF-Storel respectively because they have the same semantic meanings. The definitions are different because the MCRF has a different mechanism, compared to the CRF, for finding an address from the sache. In the MCRF, consecutive addresses are grouped and stored together in a single sacheline. The sacheline is identified by the first address of the group. Therefore, to access a particular address from the sache, the MCRF first checks whether the sacheline containing the address is cached. Then, it gets the required data at the corresponding position in the sacheline if it is cached. The expression "$(a \text{ div } n) \equiv (a' \text{ div } n)$" is used to find out the identifier of the required sacheline, while the expression "$m \equiv (a \text{ mod } n)$" calculates the position of the requested address within the sacheline.

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

MCRF-Commit Rule

$$\text{Site}(n,\text{sache},\langle t,\text{Commit}(a)\rangle;\text{pmb},\text{mpb},\text{proc}) \quad \text{if} \quad \text{Cell}(a',v_0,\text{-},v_1,\text{-},..,v_m,\text{Dirty},..) \notin \text{sache}$$
$$\vee \quad (a \text{ div } n) \not\equiv (a' \text{ div } n)$$
$$\vee \quad m \not\equiv (a \text{ mod } n)$$
$$\rightarrow \quad \text{Site}(n,\text{sache},\text{pmb},\text{mpb}|\langle t, Ack\rangle, \text{proc})$$

MCRF-Reconcile Rule

$$\text{Site}(n,\text{sache},\langle t,\text{Reconcile}(a)\rangle;\text{pmb},\text{mpb},\text{proc}) \quad \text{if} \quad \text{Cell}(a',v_0,\text{-},v_1,\text{-},..,v_m,\text{Clean},..) \notin \text{sache}$$
$$\vee \quad (a \text{ div } n) \not\equiv (a' \text{ div } n)$$
$$\vee \quad m \not\equiv (a \text{ mod } n)$$
$$\rightarrow \quad \text{Site}(n,\text{sache},\text{pmb},\text{mpb}|\langle t, Ack\rangle, \text{proc})$$

Similar to MCRF-Loadl and MCRF-Storel rules, MCRF-Commit and MCRF-Reconcile rules can also be simulated by the corresponding CRF-Commit and Commit-Reconcile rules respectively because the definitions are different only at the mechanism for finding an address from the sache.

Cache, Writeback and Purge Rules: A sache can obtain Clean copies of the addresses belonging to the same sacheline from the memory, if the sacheline is not cached at the time(thus no sache can contain more than one copy for the same address). A sacheline can be purged from the sache only if all addresses in the sacheline are Clean. A Dirty copy of an address in a sacheline can be written back to the memory individually, after which the state of the address becomes Clean. These three rules are also called the background rules, because their applications do not depend on any instruction.

MCRF-Cache Rule

$$\text{Sys}(\text{mem}, \ \text{Site}(n, \text{sache}, \text{pmb}, \text{mpb}, \text{proc}) \mid \text{sites}) \quad \text{if} \quad (a \text{ mod } n) \equiv 0$$
$$\wedge \quad \text{Cell}(a,\text{-},\text{-},..) \notin \text{sache}$$
$$\rightarrow \quad \text{Sys}(\text{mem}, \ \text{Site}(n, \text{Cell}(a,\text{mem}[a],\text{Clean},\text{mem}[a+1],\text{Clean},..,\text{mem}[a+(n-1)],\text{Clean}) \mid \text{sache}, \text{pmb}, \text{mpb}, \text{proc}) \mid \text{sites})$$

MCRF-Purge Rule

Site(n, Cell(a,-,Clean,-,Clean,..,-,Clean) | sache, pmb, mpb, proc)
$\rightarrow$ Site(n, sache, pmb, mpb, proc)

MCRF-Writeback Rule

Sys(mem, Site(n, Cell(a,v_0,c_0,v_1,c_1,..,v_m,Dirty,..) | sache, pmb, mpb, proc) | sites)
$\rightarrow$ Sys(mem[$(a+m) := v_m$], Site(n,Cell(a,v_0,c_0,v_1,c_1,..,v_m,Clean,..) | sache, pmb, mpb, proc) | sites)

Different from CRF-Cache which brings only a single address to the sache, MCRF-Cache brings a group of consecutive addresses to the sache simultaneously. The group forms a sacheline, which is identified by the first address. Therefore, by checking the sacheline identifier, the system then knows all the addresses contained in the sacheline. When the sacheline is brought to the sache, all addresses are set to Clean. MCRF-Cache can be simulated by executing CRF-Cache rule on each address of the sacheline.

MCRF-Purge allows the sache to purge the sacheline only if all addresses in the sacheline are Clean. MCRF-Purge can be simulated by executing CRF-Purge rule on each address of the sacheline. MCRF-Cache and MCRF-Purge together ensure addresses belonging to the same sacheline are always brought to or purged from a site together, which satisfies the MCRF Property 1.

Although MCRF caches and purges addresses in groups, it writes back data individually, which allows the system to write back only the dirty addresses in a sacheline. This approach avoids the correctness issue mentioned in Section 3.1. To allow addresses to be written back individually, a sacheline needs to maintain a cache state for each address of the sacheline, which explains why MCRF Property 2 is needed. MCRF-Writeback can be simulated by CRF-Writeback because they have the same semantics.

Processor Rules

Rule Name	Instruction	Cstate	Action	Next Cstate
MCRF-Loadl	Loadl(a)	Cell(a',v_0,c_0,..,v_k,Clean,..,v_{n-1},c_{n-1})*	*retire*	Cell(a',v_0,c_0,..,v_k,Clean,..,v_{n-1},c_{n-1})*
		Cell(a',v_0,c_0,..,v_k,Dirty,..,v_{n-1},c_{n-1})*	*retire*	Cell(a',v_0,c_0,..,v_k, Dirty,..,v_{n-1},c_{n-1})*
MCRF-Storel	Storel(a,v'_k)	Cell(a',v_0,c_0,..,v_k,Clean,..,v_{n-1},c_{n-1})*	*retire*	Cell(a',v_0,c_0,..,v'_k,Dirty,..,v_{n-1},c_{n-1})*
		Cell(a',v_0,c_0,..,v_k,Dirty,..,v_{n-1},c_{n-1})*	*retire*	Cell(a',v_0,c_0,..,v'_k,Dirty,..,v_{n-1},c_{n-1})*
MCRF-Commit	Commit(a)	Cell(a',v_0,c_0,..,v_k,Clean,..,v_{n-1},c_{n-1})*	*retire*	Cell(a',v_0,c_0,..,v_k,Clean,..,v_{n-1},c_{n-1})*
		$a \notin sache$	*retire*	$a \notin sache$
MCRF-Reconcile	Reconcile(a)	Cell(a',v_0,c_0,..,v_k,Dirty,..,v_{n-1},c_{n-1})*	*retire*	Cell(a',v_0,c_0,..,v_k,Dirty,..,v_{n-1},c_{n-1})*
		$a \notin sache$	*retire*	$a \notin sache$

* a' is the address identifying the sacheline which contains address a at the kth position

Background Rules

Rule Name	Cstate	Mstate	Next Cstate	Next Mstate
MCRF-Cache	$a \notin sache$ where a mod $n = 0$	Cell(a,v_0) Cell(a_1,v_1) ⋮ Cell(a_{n-1},v_{n-1})	Cell(a,v_0,c_0,..,v_{n-1},c_{n-1}) where c_0,c_1,..,c_{n-1}=Clean	Cell(a,v_0) Cell(a_1,v_1) ⋮ Cell(a_{n-1},v_{n-1})
MCRF-Writeback	Cell(a,v_0,c_0,..,v_k,Dirty, ..,v_{n-1},c_{n-1})	Cell(a,v'_0) Cell(a_1,v'_1) ⋮ Cell(a_k,v'_k) Cell(a_{k+1},v'_{k+1}) ⋮ Cell(a_{n-1},v'_{n-1})	Cell(a,v_0,c_0,..,v_k,Clean, ..,v_{n-1},c_{n-1})	Cell(a,v'_0) Cell(a_1,v'_1) ⋮ Cell(a_k,v_k) Cell(a_{k+1},v'_{k+1}) ⋮ Cell(a_{n-1},v'_{n-1})
MCRF-Purge	Cell(a,v_0,c_0,..,v_{n-1},c_{n-1}) where c_0,c_1,..,c_{n-1}=Clean	Cell(a,v'_0) Cell(a_1,v'_1) ⋮ Cell(a_{n-1},v'_{n-1})	$a \notin sache$	Cell(a,v'_0) Cell(a_1,v'_1) ⋮ Cell(a_{n-1},v'_{n-1})

Figure 3-5: Summary of the MCRF Rules

Summary of the MCRF Rules

Figure 3-5 summarizes the definitions of the MCRF rules. The rules are grouped into two categories: the processor rules and the background rules. When an instruction is completed (retired), it is removed from the processor-to-memory buffer and the corresponding data or acknowledgement is sent to the memory-to-processor buffer.

3.4 Correctness Proof of the MCRF

This section proves that the MCRF model produces answers that can also be produced by the CRF model by showing that all behaviors of the MCRF and be simulated by the CRF. The CRF can simulate the MCRF because of the following reasons: 1) Both memory models have the same set of instructions. 2) Both have the same set of rules for instruction reorderings and memory fences. 3) For other MCRF rules, each of them is proved to be simulated by the CRF rules, which the mapping is summarized in Figure 3-6. As can be seen, five rules can be simulated by a single corresponding

MCRF rule	**CRF rule**
MCRF-Loadl (a_k)	CRF-Loadl (a_k)
MCRF-Storel (a_k)	CRF-Storel (a_k)
MCRF-Commit (a_k)	CRF-Commit (a_k)
MCRF-Reconcile (a_k)	CRF-Reconcile (a_k)
MCRF-Cache (a)	CRF-Cache (a, $a+1$,...,$a+(n-1)$)
MCRF-Purge (a)	CRF-Purge (a, $a+1$,...,$a+(n-1)$)
MCRF-Writeback (a_k)	CRF-Writeback (a_k)

* a is the address identifying the sacheline

Figure 3-6: Mapping MCRF rules to CRF rules

CRF rules, while the two remaining rules can be simulated by executions of some CRF rules on multiple addresses.

Chapter 4

Multiword Base Protocol

In this chapter, we present the Multiword Base protocol, which is a protocol for the CRF memory model. Its design is based on the Base protocol [1]. The main advantage of Base is that it does not require the memory side to maintain any state. However, the focus for Base was adaptivity and correctness, it ignored some important implementation issues such as cache-line replacement, efficient buffer management and compatibility with multiword cache lines. Therefore, Multiword Base is developed to avoid these limitations. The remaining of the chapter is organized as follows:

In Section 4.1, we describe the features of the Multiword Base protocol. In Section 4.2, we present the system configurations of Multiword Base, which show all the components in Multiword Base. In Section 4.3, Section 4.4 and Section 5.3, we discuss the functionalities of the components in the cache sites, the memory site and the network respectively. The functionalities are described in rewriting rules, which form a Term Rewriting System (TRS). In Section 4.6, we prove that Multiword Base is a correct implementation of CRF by mapping the TRS of Multiword Base to the TRS of the MCRF. Moreover, we show that the system always has forward progress.

4.1 Features

The design of Multiword Base avoids some of the limitations of the original Base protocol. The followings are the features that only exist in Multiword Base:

1. **Multiword Cache Lines**. Multiword Base is compatible with multiword cache lines, which are common in commercial commodities. By maintaining a cache state for each address in a cache line and supporting fine-grain write-backs, Multiword Base is still a correct implementation of the CRF. This is proved by mapping Multiword Base to the MCRF in Section 4.6.

2. **Non-FIFO Message Passing Network**. Multiword Base allows the network to reorder messages arbitrarily. In contrast, the original Base requires the network to allow only messages with different destinations or addresses to be reordered (FIFO message passing). My approach gives more flexibility to the implementation of the network, which enhances the scalability of the system. Moreover, non-FIFO network adds the possibility of incorporating the negative acknowledgement mechanism to the protocol, which can lead to simpler buffer managements. Although Multiword Base supports non-FIFO network, it still preserves the FIFO message passing property by preventing messages with the same address, source and destination to enter the network at the same time.

3. **Simple Buffer Managements**. Multiword Base allows the incoming message buffer to be implemented as an inexpensive First In First Out (FIFO) queue. In contrast, the original Base requires the incoming message buffer to be able to reorder arbitrarily the messages with different sources or addresses. I find that this requirement is unnecessary because incoming messages in Base never block each other.

4. **Cache Replacement Policy**. The original Base ignores the possibility of cache conflicts. Therefore, it does not have any cache replacement policy. In contrast, Multiword Base specifies additional rules for cache replacements. In Section 4.6, we prove that these rules does not violate the CRF model because they can be simulated by the MCRF.

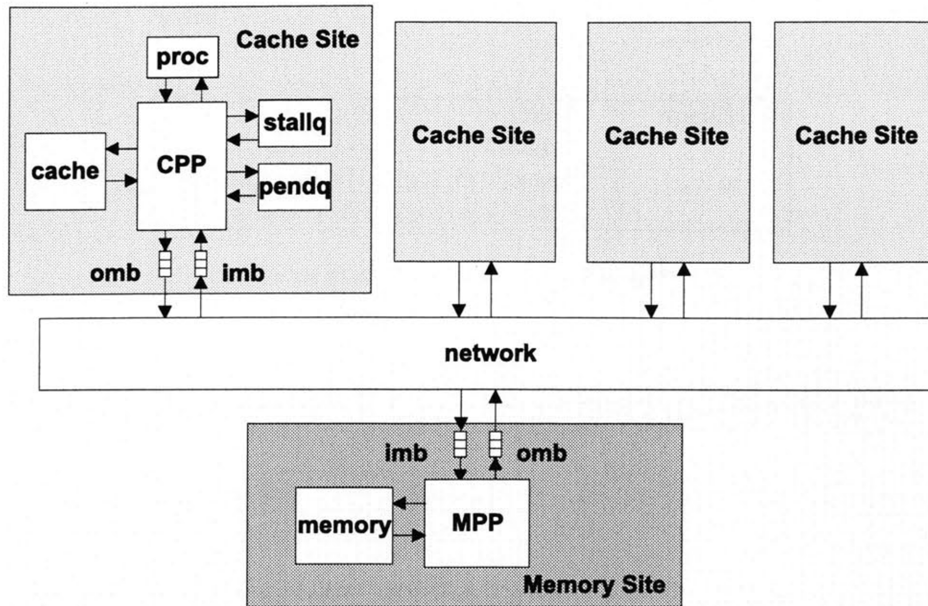


Figure 4-1: The System Configurations of Multiword Base

4.2 System Configurations

Figure 4-1 presents the system configurations of a Multiword Base system with four processors. As can be seen, the system contains four cache sites, a memory site and a network. Each cache site is identified by a unique identifier (id) and consists of a processor (proc), a cache, a pend queue (pendQ), a stall queue (stallQ), an incoming message buffer (imb), an outgoing message buffer (omb) and a cache-side protocol processor (CPP). On the other hand, the memory site consists of an incoming message buffer (imb), an outgoing message buffer (omb), the memory and a memory-side protocol processor (MPP). There is a network which connects all the cache sites and the memory sites together. The Multiword Base system executes the instruction set shown in Figure 4-2, which contains all the CRF instructions. Figure 4-3 describes the types of messages that are passed between the components. These messages contain the necessary information for the system to operate.

CRFInstr	$\equiv$	Loadl(a) \[\] Storel(a,v)
	\[\]	Commit(a) \[\] Reconcile(a)
	\[\]	$\text{Fence}_{rr}(a_1, a_2)$ \[\] $\text{Fence}_{rw}(a_1, a_2)$
	\[\]	$\text{Fence}_{wr}(a_1, a_2)$ \[\] $\text{Fence}_{ww}(a_1, a_2)$

Figure 4-2: CRF Instructions

Cache Site Messages

sender	**receiver**	**message type**	**description**
proc	CPP	⟨Tag, CRInstr⟩	*tag, CR instruction (CRF excluding fences)*
CPP	proc	⟨Tag, Result⟩	*tag, results*
CPP	cache	⟨COp, Addr, CELL⟩	*cache operation, address, cache block*
cache	CPP	⟨CAck, Addr, CELL⟩	*cache acknowledgement, address, cache block*
CPP	stallQ	⟨SOp, ⟨Tag, CRInstr⟩⟩	*stallQ operation, tag, CR instruction*
stallQ	CPP	⟨SAck, ⟨Tag, CRInstr⟩⟩	*stallQ acknowledgement, tag, CR instruction*
CPP	pendQ	⟨POp, ⟨Tag, CRInstr⟩, CELL⟩	*pendQ operation, tag, CR instruction, cache block*
pendQ	CPP	⟨PAck, ⟨Tag, CRInstr⟩, CELL⟩	*pendQ acknowledgement, tag, CR instruction, cache block*
CPP	omb	⟨MReq, Src, Dest, Addr, Data⟩	*request to memory site, source, destination, address, data*
imb	CPP	⟨MRpy, Src, Dest, Addr, Data⟩	*memory site reply, source, destination, address, data*
omb	network	⟨MReq, Src, Dest, Addr, Data⟩	*request to memory site, source, destination, address, data*
network	imb	⟨MRpy, Src, Dest, Addr, Data⟩	*memory site reply, source, destination, address, data*

Memory Site Messages

sender	**receiver**	**message type**	**description**
network	imb	⟨MReq, Src, Dest, Addr, Data⟩	*request to memory site, source, destination, address, data*
omb	network	⟨MRpy, Src, Dest, Addr, Data⟩	*memory site reply, source, destination, address, data*
imb	MPP	⟨MReq, Src, Dest, Addr, Data⟩	*request to memory site, source, destination, address, data*
MPP	omb	⟨MRpy, Src, Dest, Addr, Data⟩	*memory site reply, source, destination, address, data*
MPP	memory	⟨MOp, Addr, MBlock⟩	*memory operation, address, memory block*
memory	MPP	⟨MAck, Addr, MBlock⟩	*memory acknowledgement, address, memory block*

Figure 4-3: The Messages of Multiword Base

Brief Description of Multiword Base Operation Sequence

The follow describes the general operation sequence of the system:

1. The proc executes a stream of instructions. When it see a memory instruction, it issues the request, which is identified by a tag, to the CPP..

2. The CPP accesses the cache to check the status of the target address of the memory instruction. According to the protocol specification, it performs one of the followings:

3. (a) The CPP sends the answer to the proc.

   (b) The CPP needs to communicate with the memory site before it can answer the proc. Therefore, it sends a request to the memory site. The CPP answers the proc once it receives the reply from the memory site.

4.3 Definition of the Cache Site

In this Section, we explain the functionalities of the components of a cache site. As a reminder, cache site consists of the followings components: 1) a processor, 2) a cache, 3) a pend queue, 4) a stall queue, 5) an ongoing message buffer, 6) an incoming message buffer and 7) a cache-side protocol processor.

4.3.1 Processor

The processor (proc) is responsible for issuing memory instructions to the CPP. However, the proc does not issue Fences to CPP because they only enforce the ordering of other non-Fence instructions. The proc simply completes a Fence instruction when the Fence is ready to be issued.

Issuing Constraints of Proc

Multiword Base allows the proc to issue the memory instructions to the CPP out-of-order as long as the constraints of both data dependences and memory fences are

$I_2 \Rightarrow$ $I_1 \Downarrow$	Loadl (a')	Storel (a',v')	Fence$_{rr}$ (a'_1,a'_2)	Fence$_{rw}$ (a'_1,a'_2)	Fence$_{wr}$ (a'_1,a'_2)	Fence$_{ww}$ (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$_{rr}(a_1,a_2)$	true	true	true	true	true	true	true	$a_2 \neq a'$
Fence$_{rw}(a_1,a_2)$	true	$a_2 \neq a'$	true	true	true	true	true	true
Fence$_{wr}(a_1,a_2)$	true	true	true	true	true	true	true	$a_2 \neq a'$
Fence$_{ww}(a_1,a_2)$	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 4-4: Instruction Reordering Table

preserved. Figure 4-4 summarizes the conditions that the proc can issue two instructions out-of-order, where I_1 followed by I_2 can only be reordered if their corresponding entry in the table is evaluated "true". There is an important point needed to be clarify: if the table shows that two instructions cannot be reordered, it does not mean that the proc can issue I_2 immediately after I_1. Indeed, the proc can only issue I_2 after the completion of I_1. On the other hand, if an instruction want to be issued ahead of multiple preceding instructions that are incomplete, the proc can only do so if the instruction can be reordered with each of these instructions. Although the implementation of a proc with out-of-order issuing looks complicated, we show that it can achieved by using Reorder Buffer (ROB) in Chapter 5.

4.3.2 Cache

The cache backs up the useful data that are recently accessed by the CPP. Therefore, the CPP can retrieve those data quickly when it needs to access them again. The CPP controls the data that are stored in the cache. Figure 4-5 defines the four operations that CPP can request the cache to perform. COPMSG defines the request messages that are sent from the CPP to the cache; CACKMSG defines the messages that the cache can reply to the CPP; CELL defines the data stored in a cache line. Each cache line contains the data and the cache states of multiple (n in this case) consecutive addresses. The cache state can either be "Clean", which indicates the address has not been modified since it is stored in the cache, or "Dirty", which indicates the address has been modified at least once since it is stored in the cache; CONFLICT is assigned

COPMSG	≡	⟨Cache, a,CELL⟩
	[]	⟨Purge, a⟩
	[]	⟨Read, a⟩
	[]	⟨Update, a,CELL⟩
CACKMSG	≡	⟨CacheAck,CONFLICT, a,CELL⟩
	[]	⟨PurgeAck⟩
	[]	⟨ReadAck,HIT,CELL⟩
	[]	⟨UpdateAck,HIT⟩
CELL	≡	⟨v_0,DB, v_1,DB, .., v_{n-1},DB⟩
DB	≡	Clean [] Dirty
CONFLICT	≡	conflict [] no conflict
HIT	≡	hit [] miss

Figure 4-5: Definitions of the Cache Operations

to "conflict" when there is a cache replacement at performing "Cache" operation and HIT is assigned to "hit" when there is a cache hit. Figure 4-6 summarizes the operational semantics of the cache: 1) The "Cache" operation informs the cache to store a cache line which addresses are not currently stored in the cache. If the cache can store the cache line without replacing another cache line, it replies "no conflict" to the CPP. Otherwise, it replies "conflict" to the CPP with the information of the replaced cache line. 2) The "Purge" operation informs the cache to throw away the cache line that contains a particular address. The cache acknowledges the CPP after the completion of the request. 3) The "Read" operation informs the cache to provides the data of the cache line that contains a particular address, the cache replies "hit" with the data to the CPP if it has the cache line, otherwise, it replies "miss" if it does not have the data. 4) The "Update" operation informs the cache to update the cache line of a particular address. If the cache has the cache line, it overwrites the data and replies "hit" to the CPP. Otherwise, it performs no action and replies "miss".

4.3.3 Pend Queue

Sometimes, the CPP needs to communicate with the memory site about a cache line before it can answer a proc's request. This action can take some amount of time. During that period, it is possible that the CPP receives another request that accesses the same cache line. If this happens, the CPP needs to suspend this request until the

Message from CPP	Current State	Reply to CPP	Next State
$\langle$Cache, a, $cell\rangle$	(a,-) $\notin$ cache, no conflict	$\langle$CacheAck,no conflict,-,-$\rangle$	(a, $cell$) $\in$ cache
	(a,-) $\notin$ cache, (a', $cell'$) $\in$ cache, (a, $cell$) replaces (a', $cell'$)	$\langle$CacheAck,conflict, a', $cell'\rangle$	(a, $cell$) $\in$ cache, (a',-) $\notin$ cache
$\langle$Purge, $a\rangle$	(a,-) $\in$ cache	$\langle$PurgeAck$\rangle$	(a,-) $\notin$ cache
	(a,-) $\notin$ cache	$\langle$PurgeAck$\rangle$	(a,-) $\notin$ cache
$\langle$Read, $a\rangle$	(a, $cell$) $\in$ cache	$\langle$ReadAck,hit, $cell\rangle$	(a, $cell$) $\in$ cache
	(a,-) $\notin$ cache	$\langle$ReadAck,miss,-$\rangle$	(a,-) $\notin$ cache
$\langle$Update, a, $cell\rangle$	(a,-) $\in$ cache	$\langle$WriteAck,hit$\rangle$	(a, $cell$) $\in$ cache
	(a,-) $\notin$ cache	$\langle$WriteAck,miss$\rangle$	(a,-) $\notin$ cache

Figure 4-6: Operation Semantics of the Cache

Request from CPP	Current State	Reply to CPP	Next State
$\langle$Add, $\langle t, instr\rangle$, $cell\rangle$	$\langle$-, $cell'\rangle$ $\notin$ pendQ, pendQ not full		$\langle\langle t, instr\rangle, cell\rangle$ $\in$ pendQ
$\langle$Del, $a_k\rangle$	$\langle$-, $cell''\rangle$ $\in$ pendQ		$\langle$-, $cell''\rangle$ $\notin$ pendQ
$\langle$Get, $a_k\rangle$	$\langle$-, $cell''\rangle$ $\in$ pendQ	pendQ(a_k) = true	$\langle$-, $cell''\rangle$ $\in$ pendQ
$\langle$Get, $a_k\rangle$	$\langle$-, $cell''\rangle$ $\notin$ pendQ	pendQ(a_k) = false	$\langle$-, $cell''\rangle$ $\notin$ pendQ
	pendQ full	pendQFull = true	pendQ full
	pendQ not full	pendQFull = false	pendQ not full

* $cell$, $cell'$ are cache lines containing the same addresses
* $cell''$ contains the value of a_k at the kth position

Figure 4-7: Operations of pendQ

CPP has finished the communication with the memory site about that cache line. In this thesis, those cache lines about which the CPP is communicating with the memory site are called the pending cache lines. A mechanism is needed for distinguishing the pending cache lines from other cache lines so that instructions can be suspended correctly. The pend queue (pendQ) serves for this purpose. It keeps all the pending cache lines and the corresponding instructions that make those cache lines become pending. Therefore, the number of entries in the pendQ determines the maximum number of messages can be sent to the network at the same time.

Figure 4-7 summarizes the operations of pendQ: 1) The "add" operation inserts a pending cache line and the instruction to the pendQ providing it is not full and it has not yet had a cache line containing the same addresses. 2) The "del" operation removes an entry, identified by the address, from the pendQ. 3) The "get" operation requests pendQ to reports whether it has a cache line that contains a particular address a_k, and provides the data if it has the cache line. 4) The pendQ always tells

Request from CPP	Current State	Reply to CPP	Next State
$\langle$Add, $\langle t, instr\rangle\rangle$	$\langle t, instr\rangle \notin$ stallQ, stallQ not full		$\langle t, instr\rangle \in$ stallQ
$\langle$top$\rangle$	oldest(stallQ) = $\langle t, instr\rangle$	$\langle t, instr\rangle$	$\langle t, instr\rangle \notin$ stallQ
	stallQ full	stallQFull = true	stallQ full
	stallQ not full	stallQFull = false	stallQ not full

Figure 4-8: Operations of stallQ

the CPP whether it has free spaces. Many of the functionalities of the pendQ and the cache overlap. However, there are some major differences:

1. The pendQ is fully-associative while the cache may not be. Cache lines can be stored at any slot in the pendQ as long as there is at least one free space available.

2. The pendQ does not allow cache line to be added if it is full while the cache performs a replacement at the same situation. Therefore, the pendQ is required to tell the CPP whether it has free spaces all the time.

4.3.4 Stall Queue

As mentioned before, an instruction will be suspended if it is accessing a pending cache line. Moreover, an instruction is also suspended if the instruction requires the CPP to communicate with the memory when the pendQ is full, which indicates the CPP cannot send any more message to the network. In Multiword Base, the stall queue (stallQ) is used to keep all the suspended instructions.

Figure 4-8 summarizes the operations of the stallQ: 1) The "add" operation inserts a suspended instruction to the stallQ. 2) The "pop" operation retrieves the oldest instruction in the stallQ and then removes the instruction from the stallQ. 3) The stallQ always tells the CPP whether it has free spaces.

When the stallQ is full, the CPP will not accept any further request from the proc until it completes at least one suspended instruction from the stallQ. This approach ensures that the CPP eventually deals with the suspended instructions. Moreover, it also helps to make sure that enough resources are available for the ongoing instructions

to avoid deadlock. The stallQ can be implemented as a simple FIFO queue. The number of entries in the stallQ determines the upper bound of the number of ongoing instructions the CPP can accepts. If the stallQ is eliminated, the design becomes blocking, which allows the CPP to deal with only one instruction at a time.

4.3.5 Cache-side Incoming Message Buffer

The cache-side incoming message buffer (imb) temporarily stores the incoming reply messages that are sent from the memory when the CPP cannot deal with the incoming messages fast enough. The original Base protocol requires the imb to be able to reorder the incoming messages with different sources or addresses to avoid deadlocks. In contrast, the imb of Multiword Base does not reorder messages because the incoming messages of this protocol never block each other. The imb should have the same number of entries as pendQ to make sure the cache site never misses a reply from the memory. Otherwise, the protocol should add a negative acknowledgement mechanism to assure this property.

4.3.6 Cache-side Outgoing Message Buffer

The cache-side outgoing message buffer (omb) temporarily stores the CPP outgoing request message when the network is not fast enough to deliver the messages to the memory site. Similar to the imb, the omb does not reorder messages. The omb has the same number of entries as pendQ because the network may not be able to send a single message when the pendq is filled up.

4.3.7 Cache-side Protocol Processor

The Cache-side Protocol Processor (CPP) carries out the coherence actions of Multiword Base at cache site. Figure 4-9 and Figure 4-10 summarize the rules that the CPP executes at different situations: each row in the table represents a rule which means that the CPP will perform a particular action (Action) if it receives a message (Message Received) from a component (Source) for a particular instruction (Instruc-

Instruction	Message Received	Source	Action	
$\langle t, \langle \text{Loadl}, a_k \rangle\rangle$	$\langle t, \langle \text{Loadl}, a_k \rangle\rangle$	proc/stallQ	$\langle \text{Read}, a \rangle \rightarrow$ cache	C1
	$\langle \text{ReadAck,hit,}\langle v_0,\text{-},..,v_k,\text{Clean},..,v_{n-1},\text{-}\rangle\rangle$	cache	$\langle t, v_k \rangle \rightarrow$ proc	C2
	$\langle \text{ReadAck,hit,}\langle v_0,\text{-},..,v_k,\text{Dirty},..,v_{n-1},\text{-}\rangle\rangle$	cache	$\langle t, v_k \rangle \rightarrow$ proc	C3
	$\langle \text{ReadAck,miss,-} \rangle$ pendQFull = False *and* pendQ(a_k) = False	cache	$\langle \text{CacheReq}, a \rangle \rightarrow$ omb, $\langle \text{Add}, \langle t, \langle \text{Loadl}, a_k \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C4
	$\langle \text{ReadAck,miss,-} \rangle$ pendQFull = True *or* pendQ(a_k) = True	cache	$\langle \text{Add}, \langle t, \langle \text{Loadl}, a_k \rangle\rangle\rangle \rightarrow$ stallQ	C5
	$\langle \text{Cache},a,\langle v_0,v_1,..,v_{n-1}\rangle\rangle$	imb	$\langle t, v_k \rangle \rightarrow$ proc $\langle \text{Cache}, a, cell \rangle \rightarrow$ cache $cell = \langle v_0,\text{Clean},..,v_{n-1},\text{Clean}\rangle$	C6
	$\langle \text{CacheAck,conflict},a',cell \rangle$ $cell = \langle v_0,c_0,v_1,c_1,..,v_{n-1},c_{n-1}\rangle$ $\exists i.\ c_i = \text{Dirty}$	cache	$\langle \text{Wb}, a, \langle v_0,d_0,v_1,d_1,..,v_{n-1},d_{n-1}\rangle\rangle \rightarrow$ omb if c_i = Clean then d_i = IsClean if c_i = Dirty then d_i = IsDirty $\langle \text{Del}, \langle t, \langle \text{Loadl}, a_k \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ $\langle \text{Add}, \langle \text{-}, \langle \text{VoluntaryWB}, a' \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C7
	$\langle \text{CacheAck,conflict},a',cell \rangle$ $cell = \langle v_0,c_0,v_1,c_1,..,v_{n-1},c_{n-1}\rangle$ $\nexists i.\ c_i = \text{Dirty}$	cache	$\langle \text{Del}, \langle t, \langle \text{Loadl}, a_k \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C8
	$\langle \text{CacheAck,no conflict,-,-} \rangle$	cache	$\langle \text{Del}, \langle t, \langle \text{Loadl}, a_k \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C9
	$\langle \text{WbAck}, a' \rangle$	imb	$\langle \text{Del}, \langle \text{-}, \langle \text{VoluntaryWB}, a' \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C10
$\langle t, \langle \text{Storel}, a_k, v'_k \rangle\rangle$	$\langle t, \langle \text{Storel}, a_k, v_k \rangle\rangle$	proc/stallQ	$\langle \text{Read}, a \rangle \rightarrow$ cache	C11
	$\langle \text{ReadAck,hit,}\langle v_0,c_0,v_1,c_1,..,v_{n-1},c_{n-1}\rangle\rangle$	cache	$\langle t,\text{-} \rangle \rightarrow$ proc $\langle \text{Update}, a, cell \rangle \rightarrow$ cache $cell = \langle v_0,c_0,..,v'_k,\text{Dirty},..,v_{n-1},c_{n-1}\rangle$	C12
	$\langle \text{ReadAck,miss,-} \rangle$ pendQFull = False *and* pendQ(a_k) = False	cache	$\langle \text{CacheReq}, a \rangle \rightarrow$ omb $\langle \text{Add}, \langle t, \langle \text{Storel}, a_k, v'_k \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C13
	$\langle \text{ReadAck,miss,-} \rangle$ pendQFull = True *or* pendQ(a_k) = True	cache	$\langle \text{Add}, \langle t, \langle \text{Storel}, a_k, v'_k \rangle\rangle\rangle \rightarrow$ stallQ	C14
	$\langle \text{Cache},a,\langle v_0,v1,..,v_{n-1}\rangle\rangle$	imb	$\langle t,\text{-} \rangle \rightarrow$ proc $\langle \text{Cache}, a, cell \rangle \rightarrow$ cache $cell = \langle v_0,\text{Clean},..,v'_k,\text{Dirty},..,v_{n-1},\text{Clean}\rangle$	C15
	$\langle \text{CacheAck,conflict},a',cell \rangle$ $cell = \langle v_0,c_0,v_1,c_1,..,v_{n-1},c_{n-1}\rangle$ $\exists i.\ c_i = \text{Dirty}$	cache	$\langle \text{Wb}, a, \langle v_0,d_0,v_1,d_1,..,v_{n-1},d_{n-1}\rangle\rangle \rightarrow$ omb if c_i = Clean then d_i = IsClean if c_i = Dirty then d_i = IsDirty $\langle \text{Del}, \langle t, \langle \text{Storel}, a_k \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ $\langle \text{Add}, \langle \text{-}, \langle \text{VoluntaryWB}, a' \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C16
	$\langle \text{CacheAck,conflict},a',cell \rangle$ $cell = \langle v_0,c_0,v_1,c_1,..,v_{n-1},c_{n-1}\rangle$ $\nexists i.\ c_i = \text{Dirty}$	cache	$\langle \text{Del}, \langle t, \langle \text{Storel}, a_k \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C17
	$\langle \text{CacheAck,no conflict,-,-} \rangle$	cache	$\langle \text{Del}, \langle t, \langle \text{Storel}, a_k \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C18
	$\langle \text{UpdateAck,hit} \rangle$	cache	no action	C19
	$\langle \text{WbAck}, a' \rangle$	imb	$\langle \text{Del}, \langle \text{-}, \langle \text{VoluntaryWB}, a' \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C20

* a is the address identifying the cacheline that contains the address a_k at the kth position

Figure 4-9: CPP rules for Loadl and Storel

tion). The expression "$\langle msg \rangle \rightarrow dest$" means that the CPP sends the message "*msg*" to the component "*dest*". To simply future references, each rule is assigned an identifier (e.g. C1). The following describes the execution sequences of the CPP for completing each of the four kinds of instruction.

Execution Sequences for Loadl

First, the CPP reads the data from the cache after it has received the "Loadl" instruction from the proc or stallQ (C1). There are three possible outcomes for the read operation: 1) a "Clean" copy of the data is in the cache, 2) a "Dirty" copy of the data is in the cache and 3) the cache does not have the data. For the first two cases, the CPP replies to the proc with the data and completes the "Loadl" instruc-

Instruction	Message Received	Source	Action	
$\langle t, \langle \text{Commit}, a_k \rangle\rangle$	$\langle t, \langle \text{Commit}, a_k \rangle\rangle$	proc/stallQ	$\langle \text{Read}, a \rangle \rightarrow$ cache	C21
	$\langle \text{ReadAck,hit,} \langle v_0,\text{-},..,v_k,\text{Clean},..,v_{n-1},\text{-} \rangle\rangle$	cache	$\langle t,\text{-} \rangle \rightarrow$ proc	C22
	$\langle \text{ReadAck,hit,} \langle v_0,c_0,..,v_k,\text{Dirty},..,v_{n-1},c_{n-1} \rangle\rangle$ pendQFull = False *and* pendQ(a_k) = False	cache	$\langle \text{Wb}, a, \langle v_0,d_0,v_1,d_1,..,v_{n-1},d_{n-1} \rangle\rangle \rightarrow$ omb if c_i = Clean then d_i = IsClean if c_i = Dirty then d_i = IsDirty $\langle \text{Add}, \langle t, \langle \text{Commit}, a_k \rangle\rangle, cell \rangle \rightarrow$ pendQ $cell = \langle v_0,c_0,v_1,c_1,..,v_{n-1},c_{n-1} \rangle$ $\langle \text{Purge}, a \rangle \rightarrow$ cache	C23
	$\langle \text{ReadAck,hit,} \langle v_0,\text{-},..,v_k,\text{Dirty},..,v_{n-1},\text{-} \rangle\rangle$ pendQFull = True *or* pendQ(a_k) = True	cache	$\langle \text{Add}, \langle t, \langle \text{Commit}, a_k \rangle\rangle\rangle \rightarrow$ stallQ	C24
	$\langle \text{ReadAck,miss,-} \rangle$ pendQ(a_k) = False	cache	$\langle t,\text{-} \rangle \rightarrow$ proc	C25
	$\langle \text{ReadAck,miss,-} \rangle$ pendQ(a_k) = True	cache	$\langle \text{Add}, \langle t, \langle \text{Commit}, a_k \rangle\rangle\rangle \rightarrow$ stallQ	C26
	$\langle \text{WbAck}, a \rangle$	imb	$\langle t,\text{-} \rangle \rightarrow$ proc $\langle \text{Cache}, a, cell \rangle \rightarrow$ cache $cell = \langle v_0,\text{Clean},v_1,\text{Clean},..,v_{n-1},\text{Clean} \rangle$ $\langle\langle t, \langle \text{Commit}, a_k \rangle\rangle, cell' \rangle \in$ pendQ $cell' = \langle v_0,\text{-},v_1,\text{-},..,v_{n-1},\text{-} \rangle$	C27
	$\langle \text{CacheAck,conflict,}a',cell \rangle$ $cell = \langle v_0,c_0,v_1,c_1,..,v_{n-1},c_{n-1} \rangle$ $\exists i.\ c_i$ = Dirty	cache	$\langle \text{Wb}, a, \langle v_0,d_0,v_1,d_1,..,v_{n-1},d_{n-1} \rangle\rangle \rightarrow$ omb if c_i = Clean then d_i = IsClean if c_i = Dirty then d_i = IsDirty $\langle \text{Del}, \langle t, \langle \text{Commit}, a_k \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ $\langle \text{Add}, \langle \text{-}, \langle \text{VoluntaryWB}, a' \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C28
	$\langle \text{CacheAck,conflict,}a',cell \rangle$ $cell = \langle v_0,c_0,v_1,c_1,..,v_{n-1},c_{n-1} \rangle$ $\nexists i.\ c_i$ = Dirty	cache	$\langle \text{Del}, \langle t, \langle \text{Commit}, a_k \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C29
	$\langle \text{CacheAck,no conflict,-,-} \rangle$	cache	$\langle \text{Del}, \langle t, \langle \text{Commit}, a_k \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C30
	$\langle \text{PurgeAck} \rangle$	cache	no action	C31
	$\langle \text{WbAck}, a' \rangle$	imb	$\langle \text{Del}, \langle \text{-}, \langle \text{VoluntaryWB}, a' \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C32
$\langle t, \langle \text{Reconcile}, a_k \rangle\rangle$	$\langle t, \langle \text{Reconcile}, a_k \rangle\rangle$	proc/stallQ	$\langle \text{Read}, a \rangle \rightarrow$ cache	C33
	$\langle \text{ReadAck,hit,} \langle v_0,c_0,v_1,c_1,..,v_{n-1},c_{n-1} \rangle\rangle$ c_k = Clean $\exists i.\ c_i$ = Dirty pendQFull = False *and* pendQ(a_k) = False	cache	$\langle \text{Wb}, a, \langle v_0,d_0,v_1,d_1,..,v_{n-1},d_{n-1} \rangle\rangle \rightarrow$ omb if c_i = Clean then d_i = IsClean if c_i = Dirty then d_i = IsDirty $\langle \text{Add}, \langle t, \langle \text{Reconcile}, a_k \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ $\langle \text{Purge}, a \rangle \rightarrow$ cache	C34
	$\langle \text{ReadAck,hit,} \langle v_0,c_0,v_1,c_1,..,v_{n-1},c_{n-1} \rangle\rangle$ c_k = Clean $\exists i.\ c_i$ = Dirty pendQFull = True *or* pendQ(a_k) = True	cache	$\langle \text{Add}, \langle t, \langle \text{Commit}, a_k \rangle\rangle\rangle \rightarrow$ stallQ	C35
	$\langle \text{ReadAck,hit,} \langle v_0,c_0,v_1,c_1,..,v_{n-1},c_{n-1} \rangle\rangle$ $\forall i. c_i$ = Clean	cache	$\langle t,\text{-} \rangle \rightarrow$ proc $\langle \text{Purge}, a \rangle \rightarrow$ cache	C36
	$\langle \text{ReadAck,hit,} \langle v_0,\text{-},..,v_k,\text{Dirty},..,v_{n-1},\text{-}, \rangle\rangle$	cache	$\langle t,\text{-} \rangle \rightarrow$ proc	C37
	$\langle \text{ReadAck,miss,-} \rangle$ pendQ(a_k) = False	cache	$\langle t,\text{-} \rangle \rightarrow$ proc	C38
	$\langle \text{ReadAck,miss,-} \rangle$ pendQ(a_k) = True	cache	$\langle \text{Add}, \langle t, \langle \text{Commit}, a_k \rangle\rangle\rangle \rightarrow$ stallQ	C39
	$\langle \text{WbAck}, a \rangle$	imb	$\langle t,\text{-} \rangle \rightarrow$ proc $\langle \text{Del}, \langle \text{-}, \langle \text{Reconcile}, a \rangle\rangle,\text{-} \rangle \rightarrow$ pendQ	C40
	$\langle \text{PurgeAck} \rangle$	cache	no action	C41

* a is the address identifying the cacheline that contains the address a_k at the kth position

Figure 4-10: CPP rules for Commit and Reconcile

tion (C2, C3). For the third case, the CPP will suspend the instruction by sending it to the stallQ if the pendQ is full or has already contained an entry regarding the same cache line (C5). Otherwise, the CPP sends a "CacheReq" message to the omb and adds an entry to the pendQ (C4). The message in the omb will later be sent to the memory by the network. After the memory site completes the request, it replies to the CPP with the missing cache line. The CPP retrieves the reply from the imb, answers the proc and caches the data to the cache (C6). Again, three outcomes are possible for the cache operation: 1) no cache line is replaced, 2) a cache line with all addresses being "Clean" is replaced. 3) a cache line with at least on address being "Dirty" is replaced. For the first two cases, the CPP finished the execution for

Rule Execution Sequence
C1 → C2
C1 → C3
C1 → C5 → C1 → ...
C1 → C4 → C6 → C8
C1 → C4 → C6 → C9
C1 → C4 → C6 → C7 → C10

Figure 4-11: CPP Execution Sequences for Loadl

Loadl by deleting the entry from the pendQ (C9, C8). For the third case, the CPP writes back the cache line to the memory site and replaces the entry in the pendQ by a "VoluntaryWB" entry with the replaced cache line (C7). When the memory site completes the write-back, the CPP finish the execution by discarding the cache line and deleting the "VoluntaryWB" entry from the pendQ (C10). Figure 4-11 summarizes all the possible execution sequences for the "Loadl" instruction. The expression "rule1 → rule2" means "rule1" happens after "rule2", where "rule1" and "rule2" are the identifiers in Figure 4-9 and Figure 4-10.

Execution Sequences for Storel

First, the CPP reads the data from the cache after it has received the "Storel" instruction from the proc or stallQ (C11). There are three possible outcomes for the read operation: 1) a "Clean" copy of the data is in the cache, 2) a "Dirty" copy of the data is in the cache and 3) the cache does not have the data. For the first two cases, the CPP completes the "Storel" instruction by updating the data in the cache and sending an acknowledgement to the proc (C12 → C19). For the third case, the CPP will suspend the instruction by sending it to the stallQ if the pendQ is full or has already contained an entry regarding the same cache line (C14). Otherwise, the CPP sends a "CacheReq" message to the omb and adds an entry to the pendQ (C13). The message in the omb will later be sent to the memory by the network. After the memory site completes the request, it replies to the CPP with the missing cache line. The CPP retrieves the reply from the imb, answers the proc and caches the cache line with

Rule Execution Sequence
C11 → C12 → C19
C11 → C14 → C11 → ...
C11 → C13 → C15 → C17
C11 → C13 → C15 → C18
C11 → C13 → C15 → C16 → C20

Figure 4-12: CPP Execution Sequences for Storel

updated data (C15). Again, three outcomes are possible for the cache operation: 1) no cache line is replaced, 2) a cache line with all addresses being "Clean" is replaced. 3) a cache line with at least on address being "Dirty" is replaced. For the first two cases, the CPP finished the execution for Loadl by deleting the entry from the pendQ (C18, C17). For the third case, the CPP writes back the cache line to the memory site and replaces the entry in the pendQ by a "VoluntaryWB" entry with the replaced cache line (C16). When the memory site completes the write-back, the CPP finish the execution by discarding the cache line and deleting the "VoluntaryWB" entry from the pendQ (C20). Figure 4-12 summarizes all the possible execution sequences for the "Storel" instruction.

Execution Sequences for Commit

First, the CPP read the data from the cache after it has received the "Commit" instruction from the proc or stallQ (C21). There are three possible outcomes for the read operation: 1) a "Clean" copy of the data is in the cache, 2) a "Dirty" copy of the data is in the cache and 3) the cache does not have the data. For the first case, the CPP sends an acknowledgement to the proc and completes the "Commit" instruction (C22). For the third case, the CPP will suspend the instruction by sending it to the stallQ if the pendQ has already contained an entry regarding the same cache line (C26). Otherwise, the CPP completes the "Commit" instruction by sending an acknowledgement to the proc (C25). For the second case, the CPP needs to write back the cache line. It suspends the instruction if the pendQ is full (C24). Otherwise, it sends a "Wb" message to the omb and migrates the cache line to the pendQ (C23

Rule Execution Sequence
C21 → C22
C21 → C25
C21 → C26 → C21 → ...
C21 → C24 → C21 → ...
C21 → C23 → C31 → C27 → C29
C21 → C23 → C31 → C27 → C30
C21 → C23 → C31 → C27 → C28 → C32

Figure 4-13: CPP Execution Sequences for Commit

→ C31). The message in the omb will later be sent to the memory by the network. After the memory site completes the write-back, it acknowledges the CPP. The CPP retrieves the reply from the imb, answers the proc and migrates the cache line back to the cache (C27). Again, three outcomes are possible for the migrate (cache) operation: 1) no cache line is replaced, 2) a cache line with all addresses being "Clean" is replaced. 3) a cache line with at least on address being "Dirty" is replaced. For the first two cases, the CPP finished the execution for Loadl by deleting the entry from the pendQ (C30, C29). For the third case, the CPP writes back the cache line to the memory site and replaces the entry in the pendQ by a "VoluntaryWB" entry with the replaced cache line (C28). When the memory site completes the write-back, the CPP finish the execution by discarding the cache line and deleting the "VoluntaryWB" entry from the pendQ (C32). Figure 4-13 summarizes all the possible execution sequences for the "Commit" instruction.

Execution Sequences for Reconcile

First, the CPP reads the data from the cache after it has received the "Reconcile" instruction from the proc or stallQ (C33). There are four possible outcomes for the read operation: 1) a "Clean" copy of the data is in the cache and other addresses in the same cache line are also "Clean", 2) a "Clean" copy of the data is in the cache and at least one address in the same cache line is "Dirty", 3) a "Dirty" copy of the data is in the cache and 4) the cache does not have the data. For the first case, the CPP completes the "Reconcile" instruction by purging the cache line and acknowledging

Rule Execution Sequence
C33 → C37
C33 → C38
C33 → C36 → C41
C33 → C35 → C33 → ...
C33 → C39 → C33 → ...
C33 → C34 → C41 → C40

Figure 4-14: CPP Execution Sequences for Reconcile

the proc (C36 → C41). The third case is similar to the first case, except that the CPP needs not purge the cache line (C37). For the forth case, the CPP will suspend the instruction if the pendQ has already contained an entry regarding the same cache line (C39). Otherwise, the CPP completes the "Commit" instruction by sending an acknowledgement to the proc (C38). For the second case, the CPP needs to write back the cache line. It suspends the instruction if the pendQ is full (C24). Otherwise, it sends a "Wb" message to the omb and migrates the cache line to the pendQ (C34 → C41). The message in the omb will later be sent to the memory by the network. After the memory site completes the write-back, it acknowledges the CPP. The CPP retrieves the reply from the imb, answers the proc and discards the cache line in the pendQ (C40). Figure 4-14 summarizes all the possible execution sequences for the "Reconcile" instruction.

4.4 Definition of the Memory Site

In this section, we explain the functionalities of the components in the memory site. In Multiword Base, the memory site models a non-distributed shared memory. Non-distributed shared memories are common in Chip Multi-Processor (CMP) designs. However, for other parallel systems, distributed shared memories are more practical. RaPED allows the memory site to be replaced by a distributed shared memory with minimal modifications to the designs because the system allows messages to be delivered in arbitrary order. Figure 4.4 shows the components of the memory site: 1) a memory, 2) an imb, 3) an omb and 4) a MPP.

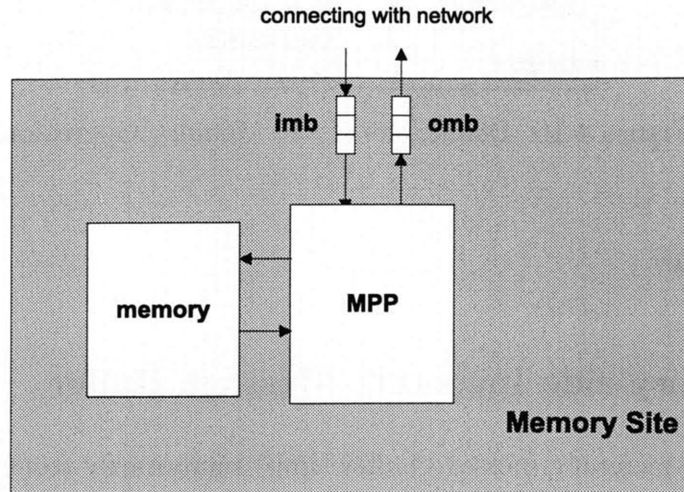


Figure 4-15: Components of the Memory Site

4.4.1 Memory

The memory stores the data of all addresses of the system. It is primarily accessed by the Memory-side Protocol Processor (MPP), which controls the data stored in the memory. Figure 4-16 defines the messages delivered between the MPP and the memory. There are two types of messages: MOPMSG and MACKMSG.

MOPMSG defines the messages delivered from the MPP to the memory. The message contains the operation that the memory is required to perform. Meanwhile, MACKMSG defines the acknowledgement messages of that the memory can reply to the MPP. Every kind of MACKMSG does not include the information about the request address because we assume the memory has fixed latency. If the assumption is not true, this information must be included in order to make Multiword Base function properly.

There are two types of operations that the memory can perform: 1) "Read" and 2) "Update". The "Read" operation returns the data of a memory row (MCELL), which contains the data of n consecutive addresses, while the "Write" operation updates the

MOPMSG	$\equiv$	$\langle$Read, $a\rangle$
	$[]$	$\langle$Update, a,MCELL$\rangle$
MACKMSG	$\equiv$	$\langle$ReadAck,MCELL$\rangle$
	$[]$	$\langle$UpdateAck$\rangle$
MCELL	$\equiv$	$\langle v_0, v_1, .., v_{n-1}\rangle$

Figure 4-16: Definitions of the Memory Operations

data of a MCELL.

4.4.2 Memory-side Incoming Message Buffer

The memory-side incoming message buffer (imb) temporarily stores the incoming requests from the cache sites when the MPP is not fast enough to deal with these requests. The original Base protocol requires the imb to be able to reorder incoming messages with different sources or addresses arbitrarily. In contrast, the imb of Multiword Base does not reorder messages because incoming messages from the cache sites of this protocol never block each other. The imb of the memory has the same number of entries as the sum of the entries in the imbs of all cache sites because this is the maximum number of messages that the MPP can receive. Since the size of the imb increases linearly with the number of processors, this approach can be expensive. A solution is to incorporates the Negative Acknowledgement (NACK) mechanism to the protocol, which requires a resend of the message when the receiver's imb does not have space. However, NACK can cause correctness issue in the original Base protocol because NACK can reorder messages with the same address, source and destination even the network passes messages in order. In contrast, NACK is compatible with Multiword Base because it always allows messages in the network to be reorder arbitrarily.

4.4.3 Memory-side Outgoing Message Buffer

The memory-side outgoing message buffer (omb) temporarily stores the MPP outgoing messages when the network cannot deliver messages to their destinations fast enough. Similar to the imb, the omb does not reorder messages. The number of en-

tries in the omb can be arbitrary. If the omb is full, the MPP stalls and then resumes after the network has sent an outgoing message from the omb.

4.4.4 Memory-side Protocol Processor

The Memory-side Protocol Processor (MPP) is responsible for handling the protocol at memory side. Figure 4-17 summarizes the operational semantics of the MPP. To simplify future reference, each row is assigned an identifier (e.g. M1). There are two types of request messages: 1) "CacheReq" messages and 2) "Wb" messages.

A cache site sends a "CacheReq" message to the MPP requesting for the data of a cache line when it has a cache miss. The message also comes with the cache site's id and the address of the cache line. When the MPP receives the "CacheReq" message, it gets the data from the memory. After getting the data, the MPP replies to the cache site with a "Cache" message which includes the data of the cache line.

A cache site sends a "Wb" message to the MPP to write back a cache line. The message also comes with the cache site's id, the address of the cache line and the write-back data. To support fine-grain write-backs, the write-back data include dirty bits, which gives the exact addresses to be updated. When the MPP receives the "Wb" message, it first gets the memory copy of the data. After getting the data from the memory, the MPP forms a new copy of the data by merging the memory copy and the write-back copy based on the dirty bits. After that, the MPP replaces the old copy in the memory by the new copy and then replies to the cache site with a "WbAck" message.

Similar to the CPP, incoming requests for the MPP never block each other because they do not require the MPP to communicate with other cache sites before the MPP returns the answer.

4.5 Definition of the Network

The network provides two communication channels between the cache sites and the memory site. One channel is responsible for delivering messages from the cache sites

Processing	Message	Sender	Action	
$\langle$CacheReq,$a\rangle$	$\langle$CacheReq,$a\rangle$	id	$\langle$Read $a\rangle$ $\rightarrow$mem	M1
	$\langle$ReadAck,a,v_0,..,$v_3\rangle$	mem	$\langle$Cache,a,v_0,..,$v_3\rangle$ $\rightarrow$id	M2
$\langle$Wb,a,v'_0,d_0,..,v'_3,$d_3\rangle$	$\langle$Wb,a,v_0,d_0,..,v_3,$d_3\rangle$	id	$\langle$Read $a\rangle$ $\rightarrow$mem	M3
	$\langle$ReadAck,a,v_0,..,$v_3\rangle$	mem	$\langle$WbAck,$a\rangle$ $\rightarrow$id $\langle$Update,a,v''_0,..,$v''_3\rangle$ $\rightarrow$mem if d_n then $v''_n = v'_n$ else $v''_n = v_n$	M4
	$\langle$UpdateAck,$a\rangle$	mem	no action	M5

Figure 4-17: MPP Rules Summary

to the memory site. Another channel is responsible for message delivery for the opposition direction. There are two requirements for the network: 1) the network is fair so that each cache site has its opportunity to deliver a message to the memory site. 2) Any message in the network can eventually reach its destination.

4.6 Correctness Proof of Multiword Base

A correct protocol for CRF should satisfies two properties: 1) soundness and 2) liveness. The former guarantees that the system does not perform any action that violates the semantics of the CRF model, while the latter assures that the system always makes forward progress. The following proves Multiword Base has both properties.

4.6.1 Soundness Proof of Multiword Base

Multiword Base is a correct protocol for the CRF because it can be simulated by the MCRF, which itself can be simulated by the CRF. The proof is done in the following steps: First, we define a mapping which maps every state in Multiword Base to a corresponding state in the MCRF. Then, we prove that each state transition of Multiword Base can be simulated by the MCRF.

Mapping Multiword Base Terms to the MCRF Terms

It is straightforward to map the terms of Multiword Base to MCRF when there is no outstanding instruction in Multiword Base (an outstanding instruction is an in-

struction that a processor has issued a request to the CPP but not yet received the reply). However, it is more difficult to map the Multiword Base with outstanding instructions to the MCRF because the two models have different mechanisms to deal with the outstanding instructions. The solution is to rewind (backward draining) or fast-forward (forward draining) the executions of outstanding instructions in Multiword Base until every outstanding instruction gets to a state which is either before the issuing or after the completion. Therefore, all states in Multiword Base can be mapped to a state in MCRF with the help of drainings.

Two kinds of draining are needed to make the process confluent. For example, if several rules lead to the same state, the backward draining cannot be used because the system does not know which rules was execute before to get to this state. Figure 4-18 depicts this scenario. On the other hand, the forward draining cannot be used when different orders of the rule executions can lead to different states. This can happen in Multiword Base when multiple cache sites trying to write back the same cache line to the memory. Since no absolute order of these write backs is guaranteed, the memory can end up with different values for the cache line. This scenario is depicted in Figure 4-19.

Simulate Multiword Base with the MCRF

Figure 4-20 maps all the possible transition sequences of Multiword Base to a corresponding MCRF sequences with the same semantic effects. In the figure, the expression "R1 $\rightarrow$ R2" means the transition "R2" is followed by the transition "R1", where "R1" and "R2" are the corresponding identifiers in Figure 4-9, Figure 4-10 and Figure 4-17. Each transition sequence has a break point. Transitions before the break point use backward draining, while transitions after the break point use forward draining. A transition sequence has more than one break points when the sequence requires the CPP to contact with the memory more than once. We separate this kind of sequences to two sequences with the first sequence marked "start VMB" in the table and the second sequence starts with the same identifier as the last identifier of the first sequence. In Multiword Base, this only happens to sequences which write

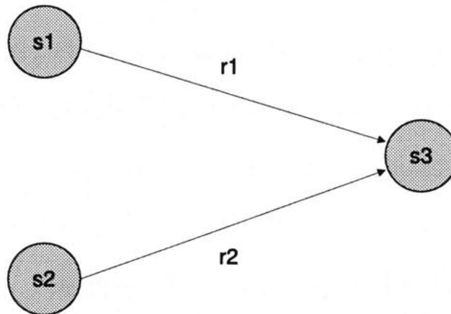


Figure 4-18: Limitation of Backward Draining (Convergence of Rules)

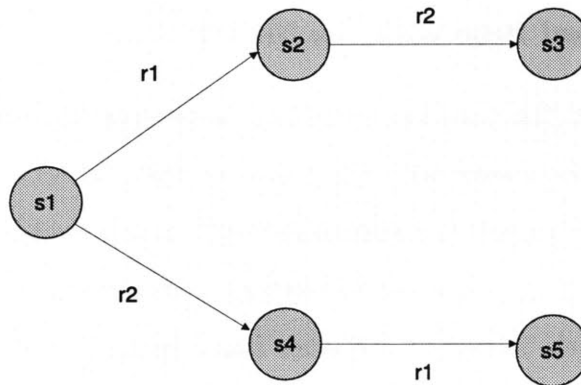


Figure 4-19: Limitation of Forward Draining (Divergence of Rules)

Instruction	Backward Draining	Forward Draining	Equivalent MCRF Semantics
Loadl a	C1 →	C2	Loadl a
	C1 →	C3	Loadl a
	C1 → C5 → C1 (loop)		no effect
	C1 → C4 →	M1 → M2 → C6 → C8	Cache a → Loadl a → Purge a'
	C1 → C4 →	M1 → M2 → C6 → C9	Cache a → Loadl a
	C1 → C4 →	M1 → M2 → C6 → C7 (start VWB)	Cache a → Loadl a
	C7 →	M3 → M4 → M5 → C10	Writeback a' → Purge a'
Storel a	C11 →	C12 → C19	Storel a
	C11 → C14 → C11 (loop)		no effect
	C11 → C13 →	M1 → M2 → C15 → C17	Cache a → Storel a → Purge a'
	C11 → C13 →	M1 → M2 → C15 → C18	Cache a →Storel a
	C11 → C13 →	M1 → M2 → C15 → C16 (start VMB)	Cache a → Storel a
	C16 →	M3 → M4 → M5 → C20	Writeback a' → Purge a'
Commit a	C21 →	C22	Commit a
	C21 →	C25	Commit a
	C21 → C26 → C21 (loop)		no effect
	C21 → C24 → C21 (loop)		no effect
	C21 → C23 →	C31 → M3 → M4 → M5 → C27 → C29	Writeback a → Commit a → Purge a'
	C21 → C23 →	C31 → M3 → M4 → M5 → C27 → C30	Writeback a → Commit a
	C21 → C23 →	C31 → M3 → M4 → M5 → C27 → C28 (start VMB)	Writeback a → Commit a
	C28 →	M3 → M4 → M5 → C32	Writeback a' → Purge a'
Reconcile a	C33 →	C37	Reconcile a
	C33 →	C38	Reconcile a
	C33 →	C36 → C41	Purge a → Reconcile a
	C33 → C35 → C33 (loop)		no effect
	C33 → C39 → C33 (loop)		no effect
	C33 → C34 →	C41 → M3 → M4 → M5 → C40	Writeback a → Purge a → Reconcile a

Figure 4-20: Mapping From Multiword Base to MCRF

backs a replaced cache line where the second sequence can always be simulated by the MCRF write-back rule.

4.6.2 Liveness Proof of Multiword Base

Multiword Base always makes forward progress if all instructions in the system can eventually be completed (i.e. a processor always gets an answer from the CPP after it has issued an instruction). We prove this happens because all transition sequences in Figure 4-20 eventually sends an answer to the CPP ($\langle t, ans \rangle \rightarrow$ proc). It is obvious that the claim is true for those sequences with no network message passing and no loop. For those sequences with messages passing to the memory site, forward progress is assured because the network is fair. Meanwhile, for those sequences with loops, they also have forward progress because the ROB processor is required to commit instructions in-order. Since the ROB can be filled up if the CPP fails to execute a

stalled instruction, the processor eventually cannot send any instruction to the CPP, which forces the CPP to execute the stalled instruction.

Chapter 5

Implementation of Multiword Base in RaPED

This chapter presents the implementation of the Multiword Base in RaPED. Figure 5-1 gives an overview to the implementation of the Multiword Base system. Multiword Base is implemented by two kinds of RaPED nodes (Processor Node and Memory Node) with a network connecting all the nodes. The processor node implements a Multiword Base's cache site and the memory node implements a Multiword Base's memory site. The following describes the implementation of each node and the network.

5.1 Processor Node

The Processor Node uses four modules of the RaPED node to implement a cache site: 1) $\textbf{Network}_{up}$, 2) **Protocol Processor**, 3) **Data Memory** and 4) $\textbf{Network}_{low}$. Figure 5-2 and Figure 5-3 summarizes the mapping of cache site components to the modules of the processor node and the definitions of the messages delivered between the modules. The implementation of each module is presented as follows:

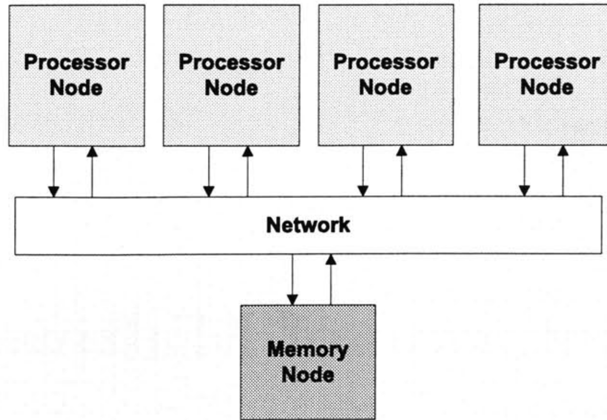


Figure 5-1: Multiword Base in RaPED

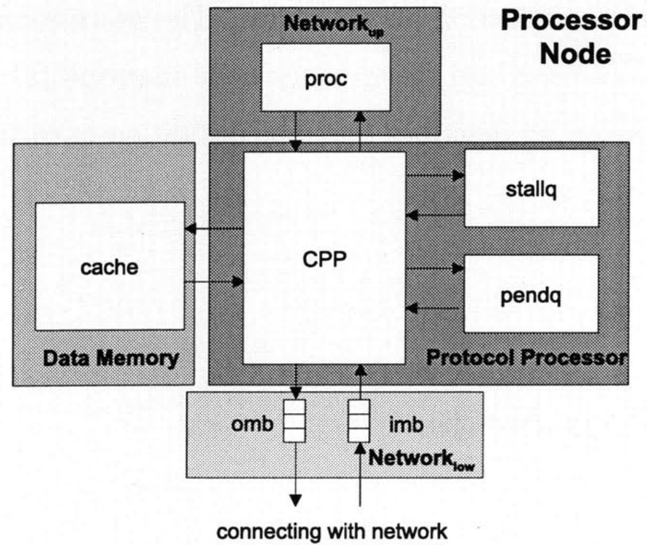


Figure 5-2: Mapping Cache Site Components to the Modules of the Processor Node

Message Type	Sender	Receiver	Message Content
$InMsg_{up}$	$Network_{up}$	Protocol Processor	$\langle$Tag, CRInstr$\rangle$
$OutMsg_{up}$	Protocol Processor	$Network_{up}$	$\langle$Tag, Result$\rangle$
DataReqMsg	Protocol Processor	Data Memory	COPMSG
DataResMsg	Data Memory	Protocol Processor	CACKMSG
$InMsg_{low}$	$Network_{low}$	Protocol Processor	$\langle$MRpy, Src, Dest, Addr, Data$\rangle$
$OutMsg_{low}$	Protocol Processor	$Network_{low}$	$\langle$MReq, Src, Dest, Addr, Data$\rangle$

Figure 5-3: Definitions of the Messages of the Processor Node

5.1.1 $\mathbf{Network}_{up}$

$\mathbf{Network}_{up}$ implements the proc of the cache site. As mentioned, Multiword Base allows the proc to issue memory instructions to the CPP out-of-order as long as the constraints of data dependences and memory fences are preserved. The implementation of the proc achieves this by having a reorder buffer (ROB). The ROB keeps all the non-committed instructions. Instructions in ROB can be in one of the following states: 1) "Not Ready", 2) "Ready To Issue", 3) "Issued" and 4) "Completed". Every instruction added to the ROB is initially marked as "Not Ready", which indicates that the proc cannot issue the instruction yet. When the ROB detects that the instruction is ready to be issued according to the Instruction Reordering Table shown in Figure 4-4, it transits the instruction to "Ready To Issue". The proc arbitrarily issues one of the "Ready To Issue" instructions and then marks the instruction as "Issued" for non-Fence instruction (or "Complete" for Fences). An "Issued" instruction becomes "Completed" when the proc receives the answer of the instruction from the CPP. "Completed" instructions are then committed in-order.

The implementation of the ROB processor in Bluespec is not presented in this thesis because Nirav Dave has already shown how to do so in his paper [7]. His processor design can be used in Multiword Base.

5.1.2 $\mathbf{Network}_{low}$

$\mathbf{Network}_{low}$ implements the imb and the omb of the cache site. Each component is implemented by a simple FIFO queue because Multiword Base does not require the messages in either component to be reordered. It is easy to implement a FIFO queue

in Bluespec because it provides the FIFO library. The following shows how the imb is implemented as a FIFO queue, which buffers message of "InMsgLow" type with n number of of entries, in Bluespec. Other components that have FIFO queues can also be implemented in a similar way.

Bluespec Code:

```
imb :: FIFO InMsgLow
imb ← mkSizedFIFO n
```

5.1.3 Data Memory

Data Memory implements the cache of the cache site. The cache is 16KB and 4-way set-associative with a single cycle latency. It has a single port for both read accesses and write accesses. The implementation of the cache is not discussed in this thesis because there are cache generators that generate Verilog code for different cache designs [12]. Bluespec programs can include components that are written in Verilog. The following example shows how to map a Verilog module to a Bluespec interface.

Bluespec Code:

```
interface Counter =
    up ::PrimAction
    preset :: Bit 4 → PrimAction
    value :: Bit 4

vCount :: Module Counter
vCount = module verilog "count4" "clk" {
    up = "enable";
    preset = "inp" "set";
    value = "outp";
}
```

The name of the Verilog module is "count4" and it is clocked by the port "clk". Moreover, its input ports: "enable", "inp" and "set" and the output port: "outp" are mapped to the Bluespec interface ("Counter") accordingly.

5.1.4 Protocol Processor

Protocol Processor contains the stallQ, the pendQ and the CPP. Their implementations are explained as follows:

PendQ

Figure 5-4 presents the implementation of the pendQ. As can be seen, there are two components in the pendQ: 1) the Freelist and 2) the Pending Slots. The Freelist keeps the pointers that point to the empty slots. Therefore, the CPP knows the pendQ is full if there is no pointer left in the Freelist. The Freelist is implemented by a simple FIFO queue. A pointer is removed (deq) from the Freelist when the CPP requests to add (enq, enq_data) an entry to the pendQ. Then, the entry is added to the Pending Slot that is pointed by that pointer. A pointer is added (enq, enq_data) back to the Freelist when the CPP removes (deq, deq_addr) a data from the pendQ. On the other hand, the Pending Slots are fully-associative. The Pending Slots and the Freelist needs to have the same number of entries. In Bluespec, the Freelist and the Pending Slots are implemented by a FIFO module and a ListN module respectively. ListN defines an abstract data type and operations for lists of a specific length. The follow presents the code for making the pendingSlots. The first line defines the pendingSlots to be a list of five elements. And Each element contains a register which holds a valid bit, an instruction message from the proc and the data of a cache line. The second line instantiates all the registers in the list with their valid bits initialized to false.

Bluespec Code:

```
pendingSlots :: ListN 5 (Reg (Valid, (Tag, CRInstr), CELL))
pendingSlots ← mapM (\c → mkReg (False,-,-)) genList
```

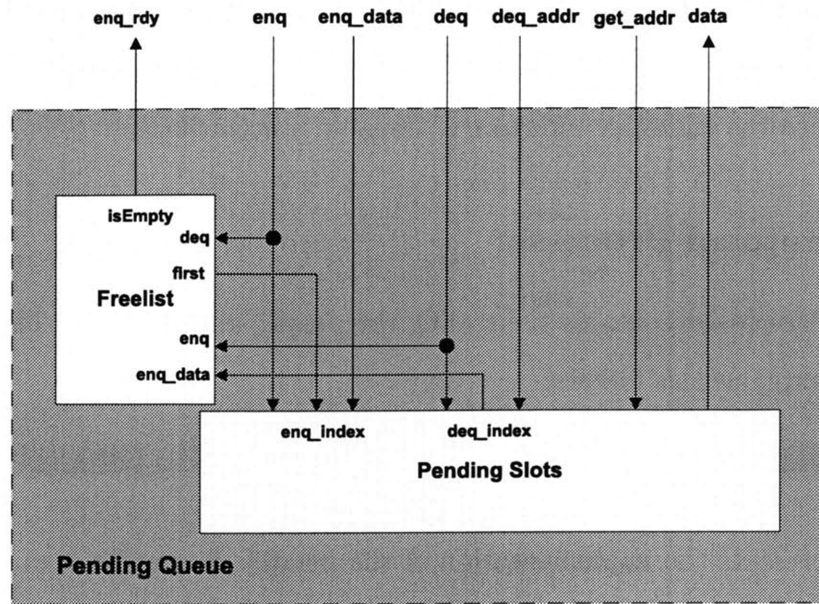


Figure 5-4: Pending Queue Implementation

StallQ

Similar to the imb and the omb, the stallQ is also implemented as a FIFO queue. However, the stallQ needs to tell the CPP whether it is full which requires the implementation of the FIFO queue to provide this information. Bluespec provides another library called FIFOF to do this. The following shows its usage which implements an n entries FIFOF:

Bluespec Code:

```
stallQ :: FIFOF (Tag, CRInstr)
stallQ ← mkSizedFIFOF n
```

CPP

The CPP consists of combinational logics that control the protocol actions at cache-side, which are summarized in Figure 4-9 and Figure 4-10. It is trivial to implement

the CPP rules from these figures in Bluespec. The following example shows how Bluespec expresses C2 from Figure 4-9. The C2 specifies that if the instruction is Loadl and the response from the cache shows that the target address is cached "Clean", then the CPP can returns the address value together with the tag to the proc.

Bluespec Code:

```
when   (instr == Loadl) &&
       (dm.getDataRes == (ReadAck Hit -))  &&
       ((getCState addr (getCELL dm.getDataRes)) == Clean) ==>
       action   nup.putOutMsgUp (tag, (getVal addr (getCELL dm.getDataRes))
```

In the example, "dm" is the **Data Memory** which implements cache and "nup" is the **Network**$_{up}$ which implements proc. Moreover, "getCELL", "getState" and "getCState" are predefined functions which extract the required information from the response message "dm.getDataRes". Bluespec defines a rule in the format of "when $c \Longrightarrow a$" where c is the execution condition and a is the action to be executed. The rule in this example can be fired when the CPP gets a read hit from the cache ("dm.getDataRes == (ReadAck Hit -)") and finds out that the cache state of the target address is Clean ("getCState *addr* (getCELL dm.getDataRes)) == Clean"). If the rule is executed, the action that will be performed is to send the result to the proc ("nup.putOutMsgUp (*tag*, (getVal *addr* (getCELL dm.getDataRes))"). All other rules of the CPP can be translated in a similar way.

5.2 Memory Node

The Memory Node uses three modules of the RaPED node to implement a memory site: 1) the **Network**$_{up}$, 2) the **Protocol Processor** and 3) the **Data Memory**. Figure 5-5 and Figure 5-6 summarizes the mapping of memory site components to the modules of the memory node and the definitions of the messages delivered between the modules. The implementation of each module of the Memory Node is presented as follows:

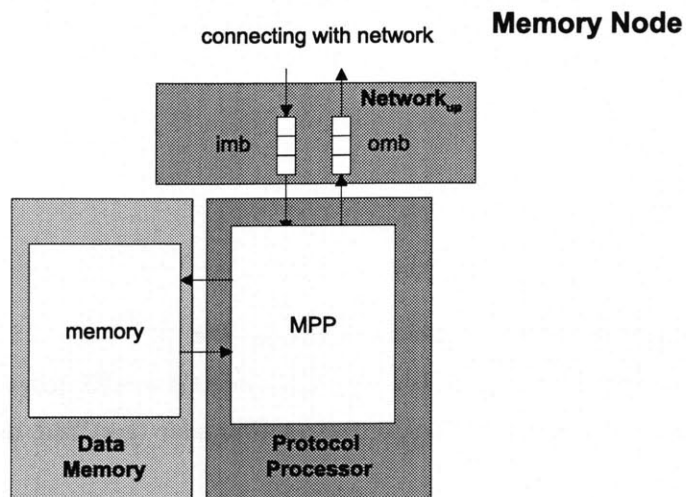


Figure 5-5: Mapping Memory Site Components to the Modules of the Memory Node

Message Type	Sender	Receiver	Message Content
InMsg$_{up}$	Network$_{up}$	Protocol Processor	⟨MReq, Src, Dest, Addr, Data⟩
OutMsg$_{up}$	Protocol Processor	Network$_{up}$	⟨MRpy, Src, Dest, Addr, Data⟩
DataReqMsg	Protocol Processor	Data Memory	MOPMSG
DataResMsg	Data Memory	Protocol Processor	MACKMSG

Figure 5-6: Definitions of the Messages of the Memory Node

5.2.1 Data Memory

Data Memory implements the shared memory of the memory site. A magic memory is implemented for Multiword Base, which pretends to have the data of all addresses by allowing multiple addresses mapped to the same entry. The memory has a fixed latency of ten cycles and is non-pipelined. The memory is 256KB with 16K rows. Each row contains 128 bits of data, which is equivalent to data of four addresses (same width as a cache line). The memory is implemented as an array of data in Bluespec. The following shows the Bluespec code that implements the magic memory:

Bluespec Code:

```
memory :: Array (Bit 14) (Bit 128)
memory ← mkArrayFull
```

The code defines the memory to be an array that has 2^{14} (16K) rows of 128-bit data. In Bluespec, an array provides two methods for user to retrieve (sub) or update (upd) its data respectively. Therefore, the array can answer the MPP's "Read" or "Update" request by calling the appropriate method. Since the magic memory only has 16K rows, it uses 14 bits (15th-28th bits of the 32-bit address) to index to the array.

5.2.2 Network$_{up}$

Network$_{up}$ implements the imb and the omb of the memory site. These components are implemented as FIFO queues. They are defined in Bluespec the same way as the imb and the omb of the cache site.

5.2.3 Protocol Processor

Protocol Processor implements the MPP. The MPP consists of combinational logics that control the protocol actions at memory-side, which are summarized in Figure 4-17. This figure can be converted to Bluespec rules the same way as the CPP.

5.3 Implementation of the Network

The network provides two communication channels between the cache sites and the memory site. One channel is responsible for delivering messages from the cache sites to the memory site. Another channel is responsible for message delivery for the opposition direction. Each channel is implemented as a simple 4-way switch. However, their operations are slightly different. Since there are multiple cache sites, the former channel needs to ensure fairness so that each cache site has chance to deliver its messages. This is achieved by an arbitrator , which gives priority to each cache site in turns to deliver its messages. The followings are the operations of the channel: 1) When the memory site is ready to accept a request, it checks which cache sites want to send a request. 2) If only one cache site has a request, it connects to that cache site and retrieves the message. 3) If more than one cache sites have requests, it connects to the cache site with the highest priority according to the arbitrator. In contrast, the other channel does not have fairness issue because there is only one memory site. The switch just connects the memory site to the destination cache site when the memory site is sending a message. Similar to the CPP and the MPP, the semantics of the network can be converted to Bluespec rules.

Chapter 6

Snoopy CRF

It is known that systems with snoopy caches [3] allow caches to get data from other caches (a.k.a. cache intervention) to reduce the miss penalty. In the original CRF, cache interventions are not allowed. Since snoopy caches are popular in system designs, it is useful if we can prove that snoopy caches can be applied to the implementation of the CRF. Therefore, we present another variant of the CRF model in this chapter: the Snoopy CRF (SCRF) model, which is the CRF model with cache interventions.

The SCRF allows cache interventions by adding several rules and maintaining some book-keeping information. The book-keeping information helps to decompose the Clean state into two types: Clean_{sync} and $\text{Clean}_{nonsync}$. When an address is Clean_{sync}, it indicates that the copy in the sache has the same value as the copy in the memory. Therefore, if another sache needs the data, either the sache or the memory can provide the data. On the other hand, the SCRF always allows a sache to provide the data if it has a Dirty copy.

This chapter is organized as followed: In Section 6.1, we discuss the instructions and the system configurations of the SCRF. In Section 6.2, we define the rules of the SCRF. Finally, in Section 6.3, we prove that the SCRF can be simulated by the CRF, which means that a protocol for the SCRF automatically converts to a protocol for the CRF.

6.1 SCRF Instructions and System Configurations

Figure 6-1 presents the instructions and system configurations of the SCRF. The SCRF and the CRF share the same set of instructions: Loadl, Storel, Reconcile, Commit, Fence_{rr},

Fence_{rw}, Fence_{wr}, Fence_{ww}. In the SCRF, a system contains a shared memory and a list of sites. Each site is composed of a processor (proc), a processor-to-memory buffer (pmb), a memory-to-processor buffer (mpb) and a semantic cache (sache). The proc is responsible for sending SCRF instructions to the pmb. The pmb buffers the messages delivered from the proc to the sache. Messages in pmb can be reordered unless there are data dependences or memory fences. On the other hand, the mpb buffers the results of the memory operations delivered from the sache to the proc. In contrast to the pmb, messages in the mpb can always be reordered arbitrarily. Each site is connected to the shared memory where the memory is used as the data rendezvous of the system. As seen in the figure, there is a communication channel connecting all the saches together. This channel is used for cache snoops. We can also see the definition of CSTATE has two types of Clean state: Clean_{sync}, $\text{Clean}_{nonsync}$. The former means that the value in the sacheline is the same as that in the memory. The latter indicates the opposite. This classification allows cache interventions if the sache has a updated clean copy.

6.2 Rewrite Rules of the SCRF

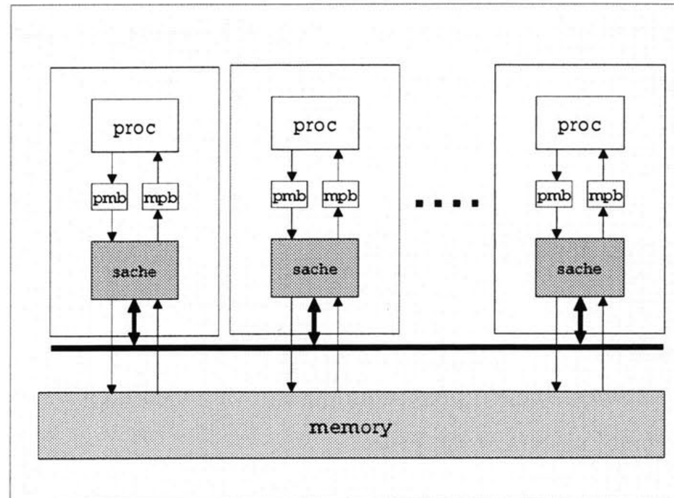
Same as the CRF, the SCRF has 2 sets of rules: The first set defines the operational semantics of Loadl, Storel, Commit and Reconcile instructions together with background rules that govern data propagation between saches and the memory. Meanwhile, the second set defines the semantics of instruction reorderings and memory fences. The SCRF rules only differ from the CRF rules by the first set. The two models share the same definition for the second set. Therefore, we only present the definitions of the first set of rules in this thesis. We also discuss how each rule in the first set can be simulated by the original CRF rules. For reference, we have included the definition of the original CRF model in Appendix A.

Loadl and Storel Rules: A Loadl or Storel can be performed if the cell containing the address is cached in the sache. A Loadl instruction returns the value in the cell to the processor through memory processor buffer (mpb). A Storel instruction updates the value in the cell accordingly and then acknowledges the processor through mpb.

SCRF Instructions

INST	$\equiv$	Loadl(a) ▯ Storel(a,v)
	▯	Commit(a) ▯ Reconcile(a)
	▯	$\text{Fence}_{rr}(a1, a2)$ ▯ $\text{Fence}_{rw}(a1, a2)$
	▯	$\text{Fence}_{wr}(a1, a2)$ ▯ $\text{Fence}_{ww}(a1, a2)$

SCRF System Configurations



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_{sync} ▯ $\text{Clean}_{nonsync}$ ▯ Dirty	*Cache State*
PMB	$\equiv$	ϵ ▯ ⟨t,INST⟩;PMB	*Processor-to-Memory Buffer*
MPB	$\equiv$	ϵ ▯ ⟨t,REPLY⟩ | PMB	*Memory-to-Processor Buffer*
REPLY	$\equiv$	v ▯ Ack	*Reply*

Figure 6-1: Instructions and System Configurations of the SCRF

SCRF-Loadl Rule

Site(sache, $\langle t$, Loadl$(a)\rangle$; pmb, mpb, proc) if Cell(a, v,-) $\in$ sache

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

SCRF-Storel Rule

Site(Cell(a,-,-)| sache, $\langle t$, Storel$(a, v)\rangle$; pmb, mpb, proc)

$\rightarrow$ Site(n, Cell(a,v,Dirty)| sache, pmb, mpb$|\langle t, Ack\rangle$, proc)

The SCRF-Loadl and the SCRF-Storel can be simulated by the CRF-Loadl and the CRF-Storel respectively because they have the same definitions.

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

SCRF-Commit Rule

Site(sache, $\langle t$,Commit$(a)\rangle$;pmb, mpb, proc) if Cell(a,v,Dirty) $\notin$ sache

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

SCRF-Reconcile Rule

Site(sache, $\langle t$,Reconcile$(a)\rangle$;pmb, mpb, proc) if Cell(a,v,Clean) $\notin$ sache

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

Similar to the SCRF-Loadl and the SCRF-Storel , the SCRF-Commit and the SCRF-Reconcile have the same semantics as the corresponding rules of the CRF.

Cache, Writeback and Purge Rules: A sache can obtain a Clean_{sync} copy of an address from the memory, if the address has not yet been cached. A Dirty copy of an address can be written back to the memory, after which the sache state of that address in the sacheline becomes Clean_{sync}. An address can be purged from the sache if its cache state is Clean_{sync} or $\text{Clean}_{nonsync}$.

SCRF-Cache Rule

Sys(mem, Site(sache, pmb, mpb, proc) | sites) if Cell(a,-,-) $\notin$ sache

$\rightarrow$ Sys(mem, Site(Cell(a,mem[a],Clean_{sync})| sache, pmb, mpb, proc) | sites)

SCRF-Writeback Rule

Sys(mem, Site(Cell(a,v,Dirty) | sache, pmb, mpb, proc) | sites)

$\rightarrow$ Sys(mem[$a := v$], Site(Cell(a,v,Clean_{sync}) | sache, pmb, mpb, proc) | sites′)

where sites′$\equiv$ sites with all Cell(a,-,Clean_{sync}) in it changed to Cell(a,-,$\text{Clean}_{nonsync}$)

SCRF-Purge Rule

Site(Cell(a,-,cstate)| sache, pmb, mpb, proc) if cstate $\equiv \text{Clean}_{sync}$

$\vee$ cstate $\equiv \text{Clean}_{nonsync}$

$\rightarrow$ Site(sache, pmb, mpb, proc)

The SCRF-Cache brings an uncached address to the site from the memory. The cache state of the address is set as Clean_{sync} because the data of the address is up-to-date with the copy in the memory. The SCRF-Cache and the CRF-Cache are different because the SCRF needs to maintain extra book-keeping information. However the information affects only the performance but not the results of the SCRF. Therefore, the SCRF-Cache can be simulated by the CRF-Cache.

The SCRF-Writeback allows a site to update a dirty copy of an address to the memory. After the writeback, the SCRF sets the cache state to Clean_{sync} because the sache and the memory now have the same data for the address. The rule also notices all other sites about the writeback, so that they can change the cache state from Clean_{sync} to $\text{Clean}_{nonsync}$. The SCRF-Writeback is nearly identical to the CRF-Writeback except the SCRF needs to inform other sites about the writeback. Similar to the SCRF-Cache, the notifications does not affect the results of the SCRF. Therefore, the SCRF-Writeback can be simulated by the CRF-Writeback.

The SCRF-Purge has the same semantic as the CRF-Purge, which allows a site to purge a copy if the copy is in one of the Clean states (either Clean_{sync} or $\text{Clean}_{nonsync}$.

Clean-Intervention and Dirty-Intervention Rules: A sache can provide the address to another sache if the address is cached at Clean_{sync} or Dirty state. The latter also requires the sache to writeback the address to the memory.

SCRF-Clean-Intervention Rule

Sys(mem, Site(sache, pmb, mpb, proc) | Site(sache′, pmb′, mpb′, proc′) | sites)

if Cell(a,-,-) $\notin$ sache $\wedge$ Cell(a,v,Clean_{sync})$\in$ sache′

$\rightarrow$ Sys(mem, Site(Cell(a,v,Clean_{sync})| sache, pmb, mpb, proc) |
Site(sache′, pmb′, mpb′, proc′) | sites)

SCRF-Dirty-Intervention Rule

Sys(mem, Site(sache, pmb, mpb, proc) |
Site(Cell(a,v,Dirty) | sache′, pmb′, mpb′, proc′) | sites)

if Cell(a,-,-) $\notin$ sache

$\rightarrow$ Sys(mem[$a := v$], Site(Cell(a,v,Clean_{sync}) | sache, pmb, mpb, proc) |
Site(Cell(a,v,Clean_{sync}) | sache′, pmb′, mpb′, proc′) | sites′)

where sites′$\equiv$ sites with all Cell(a,v',Clean_{sync}) in it changed to Cell(a,v',$\text{Clean}_{nonsync}$)

The SCRF has two additional rules related to Intervention: SCRF-Clean-Intervention and SCRF-Dirty-Intervention. The SCRF-Clean-Intervention allows a sache to provide the data of an address to another sache if the address is cached at Clean_{sync} state. This rule can be simulated by the CRF-Cache rule because the Clean_{sync} value of the copy in the sache and the value of the copy are the same (implied by Clean_{sync}). After the operation, the cache state of the address at the destination is set to Clean_{sync}.

The SCRF-Dirty-Intervention allows a sache to provide an address to another sache if the address is cached at Dirty state. The rule also requires the sache to write back the address value to the memory. This rule can be simulated by the CRF-Writeback followed by the CRF-Cache.

SCRF Rules Summary

Figure 6-2 summarizes the definitions of the SCRF rules. The rules are grouped into two categories: the processor rules and the background rules. When an instruction is completed (retired), it is removed from the processor-to-memory buffer and the corresponding data or acknowledgement is sent to the memory-to-processor buffer.

Processor Rules

Rule Name	Instruction	Cstate	Action	Next Cstate
SCRF-Loadl	Loadl(a)	Cell(a,v,Clean$_{sync}$)	*retire*	Cell(a,v,Clean$_{sync}$)
		Cell(a,v,Clean$_{nonsync}$)	*retire*	Cell(a,v,Clean$_{nonsync}$)
		Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)
SCRF-Storel	Storel(a,v)	Cell(a,-,Clean$_{sync}$)	*retire*	Cell(a,v,Dirty)
		Cell(a,-,Clean$_{nonsync}$)	*retire*	Cell(a,v,Dirty)
		Cell(a,-,Dirty)	*retire*	Cell(a,v,Dirty)
SCRF-Commit	Commit(a)	Cell(a,v,Clean$_{sync}$)	*retire*	Cell(a,v,Clean$_{sync}$)
		Cell(a,v,Clean$_{nonsync}$)	*retire*	Cell(a,v,Clean$_{nonsync}$)
		$a \notin sache$	*retire*	$a \notin sache$
SCRF-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
SCRF-Cache	$a \notin sache$	Cell(a,v)	Cell(a,v,Clean$_{sync}$)	Cell(a,v)
SCRF-Writeback	local: Cell(a,v,Dirty) remote: Cell(a,v',Clean$_{sync}$)	Cell(a,v')	local: Cell(a,v,Clean$_{sync}$) remote: Cell(a,v',Clean$_{nonsync}$)	Cell(a,v)
SCRF-Purge	Cell(a,-,Clean$_{sync}$)	Cell(a,v)	$a \notin sache$	Cell(a,v)
	Cell(a,-,Clean$_{nonsync}$)	Cell(a,v)	$a \notin sache$	Cell(a,v)
SCRF-Clean-Intervention	local: $a \notin sache$ remote: Cell(a,v,Clean$_{sync}$)	Cell(a,v)	local: Cell(a,v,Clean$_{sync}$) remote: Cell(a,v,Clean$_{sync}$)	Cell(a,v)
SCRF-Dirty-Intervention	local: $a \notin sache$ remote: Cell(a,v,Dirty) remote: Cell(a,v',Clean$_{sync}$)	Cell(a,v')	local: Cell(a,v,Clean$_{sync}$) remote: Cell(a,v,Clean$_{sync}$) remote: Cell(a,v',Clean$_{nonsync}$)	Cell(a,v)

Figure 6-2: Summary of the SCRF Rules

6.3 Proof of the Correctness of the SCRF model

This section proves that the SCRF model produces answers that can also be produced by the CRF model by showing that all behaviors of the SCRF can be simulated by the CRF. The CRF can simulate the CRF because of the following reasons: 1) Both memory models have the same set of instructions. 2) Both share the same set of rules for instruction reorderings and memory fences. 3) For other SCRF rules, they are each proved to be able to be simulated by some combination of the CRF, which are summarized by Figure 6-3. As can be seen, seven rules of SCRF can be simulated by the corresponding CRF rules. On the other hand, the CRF can also simulate the two additional intervention rules: the SCRF-Clean-Intervention and the SCRF-Dirty-Intervention. The former can be simulated by the CRF-Cache and the latter can be simulated by CRF-Writeback followed by the CRF-Cache.

SCRF rule	**CRF rule**
SCRF-Loadl (a)	CRF-Loadl (a)
SCRF-Storel (a)	CRF-Storel (a)
SCRF-Commit (a)	CRF-Commit (a)
SCRF-Reconcile (a)	CRF-Reconcile (a)
SCRF-Cache (a)	CRF-Cache (a)
SCRF-Purge (a)	CRF-Purge (a)
SCRF-Writeback (a)	CRF-Writeback (a)
SCRF-Clean-Intervention (a)	CRF-Cache (a)
SCRF-Dirty-Intervention (a)	CRF-Writeback (a) then CRF-Cache(a)

Figure 6-3: Mapping SCRF rules to CRF rules

Chapter 7

Summary and Conclusions

Framework for Rapid Protocol Engine Development

In this thesis, we first present the framework for Rapid Protocol Engine Development (RaPED). The RaPED distributes the implementation of a cache coherence protocol engine into nodes and generalizes the design of the node. The RaPED allows designs to be highly modular, which enhances the flexibility of implementation of different designs. Figure 7-1 presents the overview of a RaPED node. As can be seen, each node can be divided into five modules. Each module is responsible for a distinct functionality: 1) Data Memory is responsible for storing data, 2) Protocol Processor is responsible for carrying out the coherence actions according to the protocol specification, 3) Network_{up} is responsible for communication with the nodes towards the processor, 4) Network_{low} is responsible for communication with the nodes toward the shared memory and 5) Network_{peer} is responsible for communication with the nodes at the same level.

Multiword CRF

In the second part of the thesis, we present a variant of the Commit-Reconcile Fences (CRF) memory model called Multiword CRF (MCRF). The MCRF is optimized for CRF implementations with multiword wide cache lines. All possible execution behaviors of the MCRF model can be simulated by the CRF model. Therefore, a correct implementation of the MCRF automatically converts to a correct implementation of the CRF. There are two main modifications in the MCRF so that it is compatible with multiword cache lines

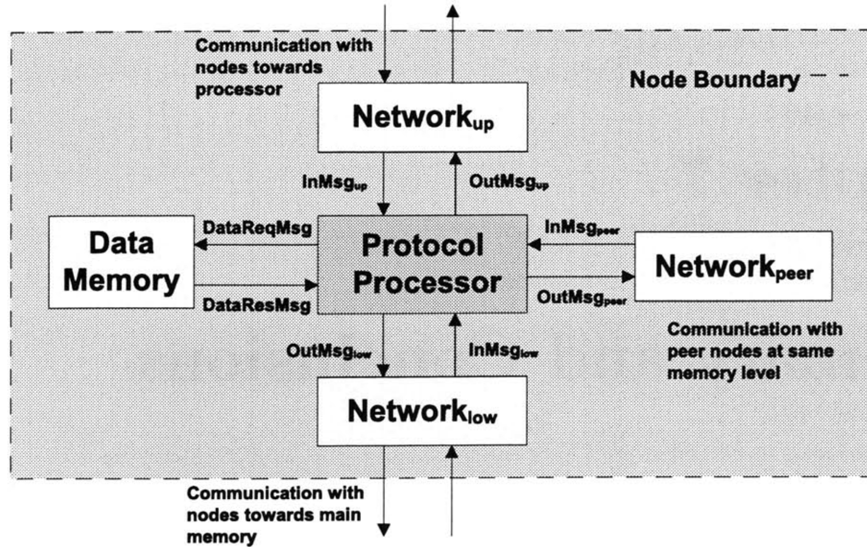


Figure 7-1: Overview of RaPED

without violating the semantics of the CRF model: 1) Addresses belonging to the same cache line are cached and purged in group. 2) Each cache line maintains a cache state for each address in the cache line, which allows the system to write back dirty address in the cache line individually. Figure 7-2 summarizes the definitions of the MCRF rules. As can be seen, most MCRF rules except the MCRF-Cache and the MCRF-Purge can be simulated by the corresponding CRF rules. While the remaining two rules can be simulated by executing the corresponding CRF rule once for each address a the cache line. This mapping is summarized by Figure 7-3. The MCRF is used to prove the correctness of Multiword Base Protocol introduced in the third part of the thesis.

Multiword Base Protocol

In the third part of the thesis, we use the RaPED to implement the Multiword Base Protocol, which is an implementation of the MCRF. Since the MCRF itself can be simulated by the CRF, the implementation is also a correct implementation of the CRF. The Multiword Base Protocol is based on the Base protocol [1]. However, the focus for Base was adaptivity and correctness, it ignored some important implementation issues such as cache-

Processor Rules

Rule Name	Instruction	Cstate	Action	Next Cstate
MCRF-Loadl	Loadl(a)	Cell(a',v_0,c_0,..,v_k,Clean,..,v_{n-1},c_{n-1})*	*retire*	Cell(a',v_0,c_0,..,v_k,Clean,..,v_{n-1},c_{n-1})*
		Cell(a',v_0,c_0,..,v_k,Dirty,..,v_{n-1},c_{n-1})*	*retire*	Cell(a',v_0,c_0,..,v_k, Dirty,..,v_{n-1},c_{n-1})*
MCRF-Storel	Storel(a,v'_k)	Cell(a',v_0,c_0,..,v_k,Clean,..,v_{n-1},c_{n-1})*	*retire*	Cell(a',v_0,c_0,..,v'_k,Dirty,..,v_{n-1},c_{n-1})*
		Cell(a',v_0,c_0,..,v_k,Dirty,..,v_{n-1},c_{n-1})*	*retire*	Cell(a',v_0,c_0,..,v'_k,Dirty,..,v_{n-1},c_{n-1})*
MCRF-Commit	Commit(a)	Cell(a',v_0,c_0,..,v_k,Clean,..,v_{n-1},c_{n-1})*	*retire*	Cell(a',v_0,c_0,..,v_k,Clean,..,v_{n-1},c_{n-1})*
		$a \notin sache$	*retire*	$a \notin sache$
MCRF-Reconcile	Reconcile(a)	Cell(a',v_0,c_0,..,v_k,Dirty,..,v_{n-1},c_{n-1})*	*retire*	Cell(a',v_0,c_0,..,v_k,Dirty,..,v_{n-1},c_{n-1})*
		$a \notin sache$	*retire*	$a \notin sache$

* a' is the address identifying the sacheline which contains address a at the kth position

Background Rules

Rule Name	Cstate	Mstate	Next Cstate	Next Mstate
MCRF-Cache	$a \notin sache$ where a mod $n = 0$	Cell(a,v_0) Cell(a_1,v_1) ⋮ Cell(a_{n-1},v_{n-1})	Cell(a,v_0,c_0,..,v_{n-1},c_{n-1}) where c_0,c_1,..,c_{n-1}=Clean	Cell(a,v_0) Cell(a_1,v_1) ⋮ Cell(a_{n-1},v_{n-1})
MCRF-Writeback	Cell(a,v_0,c_0,..,v_k,Dirty, ..,v_{n-1},c_{n-1})	Cell(a,v'_0) Cell(a_1,v'_1) ⋮ Cell(a_k,v'_k) Cell(a_{k+1},v'_{k+1}) ⋮ Cell(a_{n-1},v'_{n-1})	Cell(a,v_0,c_0,..,v_k,Clean, ..,v_{n-1},c_{n-1})	Cell(a,v'_0) Cell(a_1,v'_1) ⋮ Cell(a_k,v_k) Cell(a_{k+1},v'_{k+1}) ⋮ Cell(a_{n-1},v'_{n-1})
MCRF-Purge	Cell(a,v_0,c_0,..,v_{n-1},c_{n-1}) where c_0,c_1,..,c_{n-1}=Clean	Cell(a,v'_0) Cell(a_1,v'_1) ⋮ Cell(a_{n-1},v'_{n-1})	$a \notin sache$	Cell(a,v'_0) Cell(a_1,v'_1) ⋮ Cell(a_{n-1},v'_{n-1})

Figure 7-2: Summary of the MCRF Rules

MCRF rule	CRF rule
MCRF-Loadl (a_k)	CRF-Loadl (a_k)
MCRF-Storel (a_k)	CRF-Storel (a_k)
MCRF-Commit (a_k)	CRF-Commit (a_k)
MCRF-Reconcile (a_k)	CRF-Reconcile (a_k)
MCRF-Cache (a)	CRF-Cache (a, $a+1$,...,$a+(n-1)$)
MCRF-Purge (a)	CRF-Purge (a, $a+1$,...,$a+(n-1)$)
MCRF-Writeback (a_k)	CRF-Writeback (a_k)

* a is the address identifying the sacheline

Figure 7-3: Mapping MCRF rules to CRF rules

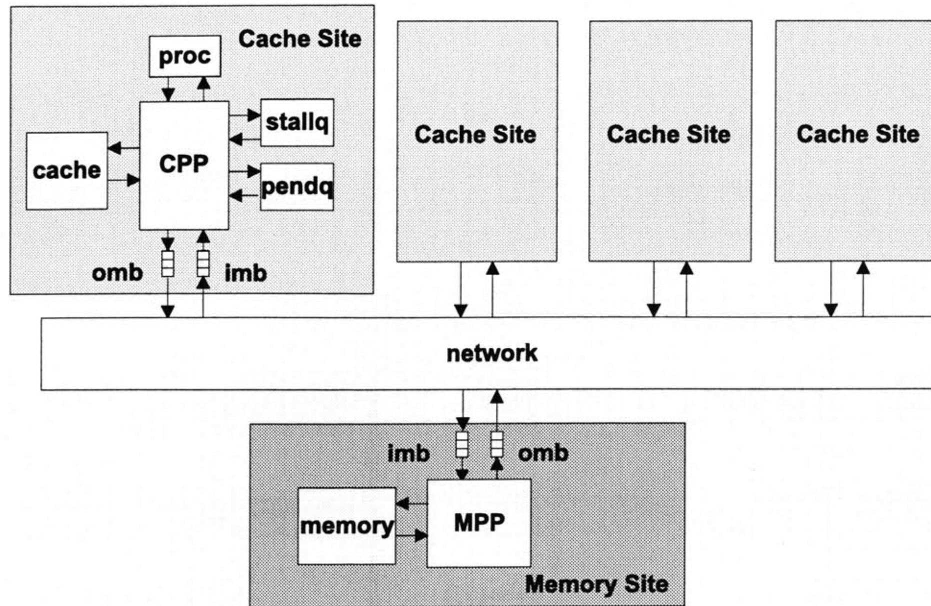


Figure 7-4: The Overview of the Implementation of Multiword Base

line replacement, efficient buffer management and compatibility with multiword cache lines. Therefore, Multiword Base is developed to avoid these limitations.

Figure 7-4 gives an overview of the implementation of Multiword Base. It shows all the components in Multiword Base. We prove that Multiword Base can be simulated by the MCRF. Since the MCRF itself can be simulated the CRF, Multiword Base is also a correct implementation of the CRF. Moreover, we prove that the Multiword Base always has forward progress.

Snoopy CRF

In the final part of the thesis, we present another variant of the Commit-Reconcile Fences (CRF) memory model called Snoopy CRF (SCRF). The SCRF is optimized for CRF implementations with multiword wide cache lines. All possible execution behaviors of the SCRF model can be simulated by the CRF model. Therefore, a correct implementation of the SCRF automatically converts to a correct implementation of the CRF. The SCRF adds some rules and some extra book keepings to allow cache interventions. The book keeping

Processor Rules

Rule Name	Instruction	Cstate	Action	Next Cstate
SCRF-Loadl	Loadl(a)	Cell(a,v,Clean_{sync})	*retire*	Cell(a,v,Clean_{sync})
		Cell(a,v,$\text{Clean}_{nonsync}$)	*retire*	Cell(a,v,$\text{Clean}_{nonsync}$)
		Cell(a,v,Dirty)	*retire*	Cell(a,v,Dirty)
SCRF-Storel	Storel(a,v)	Cell(a,-,Clean_{sync})	*retire*	Cell(a,v,Dirty)
		Cell(a,-,$\text{Clean}_{nonsync}$)	*retire*	Cell(a,v,Dirty)
		Cell(a,-,Dirty)	*retire*	Cell(a,v,Dirty)
SCRF-Commit	Commit(a)	Cell(a,v,Clean_{sync})	*retire*	Cell(a,v,Clean_{sync})
		Cell(a,v,$\text{Clean}_{nonsync}$)	*retire*	Cell(a,v,$\text{Clean}_{nonsync}$)
		$a \notin sache$	*retire*	$a \notin sache$
SCRF-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
SCRF-Cache	$a \notin sache$	Cell(a,v)	Cell(a,v,Clean_{sync})	Cell(a,v)
SCRF-Writeback	local: Cell(a,v,Dirty) remote: Cell(a,v',Clean_{sync})	Cell(a,v')	local: Cell(a,v,Clean_{sync}) remote: Cell(a,v',$\text{Clean}_{nonsync}$)	Cell(a,v)
SCRF-Purge	Cell(a,-,Clean_{sync})	Cell(a,v)	$a \notin sache$	Cell(a,v)
	Cell(a,-,$\text{Clean}_{nonsync}$)	Cell(a,v)	$a \notin sache$	Cell(a,v)
SCRF-Clean-Intervention	local: $a \notin sache$ remote: Cell(a,v,Clean_{sync})	Cell(a,v)	local: Cell(a,v,Clean_{sync}) remote: Cell(a,v,Clean_{sync})	Cell(a,v)
SCRF-Dirty-Intervention	local: $a \notin sache$ remote: Cell(a,v,Dirty) remote: Cell(a,v',Clean_{sync})	Cell(a,v')	local: Cell(a,v,Clean_{sync}) remote: Cell(a,v,Clean_{sync}) remote: Cell(a,v',$\text{Clean}_{nonsync}$)	Cell(a,v)

Figure 7-5: Summary of the SCRF Rules

SCRF rule	CRF rule
SCRF-Loadl (a)	CRF-Loadl (a)
SCRF-Storel (a)	CRF-Storel (a)
SCRF-Commit (a)	CRF-Commit (a)
SCRF-Reconcile (a)	CRF-Reconcile (a)
SCRF-Cache (a)	CRF-Cache (a)
SCRF-Purge (a)	CRF-Purge (a)
SCRF-Writeback (a)	CRF-Writeback (a)
SCRF-Clean-Intervention (a)	CRF-Cache (a)
SCRF-Dirty-Intervention (a)	CRF-Writeback (a) then CRF-Cache(a)

Figure 7-6: Mapping SCRF rules to CRF rules

information allows the SCRF to classify the Clean state into two categories: Clean_{sync} and $\text{Clean}_{nonsync}$. When an address cached in the sache is in Clean_{sync}, it indicates that the copy in the sache has the same value as the copy in the memory. Therefore, if another sache needs the data, it shows no difference between providing data from the sache and providing from the memory. On the other hand, the SCRF always allows a sache to provide the data if it has a Dirty copy. Figure 7-5 summarizes the definitions of the SCRF rules. As can be seen, there are two additional rules explicitly for cache intervention. Other SCRF rules are nearly identical to their corresponding CRF rules except the SCRF rules have some extra book keeping actions. Since these book keepings do not affect the results of the SCRF execution, these SCRF rules can be simulated by the corresponding CRF rules. Figure 7-6 shows how each SCRF rule can be simulated by CRF rules. As can be seen, the two intervention rules can also be simulated by the CRF.

7.1 Future Work

More Implementations with RaPED

It is worth testing the designs of the RaPED by using it to implement other cache coherence protocol designs. This helps to improve the framework. For example, the data memory interface assumes the module has fixed latency. However, non-uniform cache architectures (NUCA) [10, 11], which have different latencies at accessing different locations of the cache, are getting popular in the computer designs. Therefore, we may need to modify the RaPED so that NUCA cache can be implemented by the data memory.

Combination of MCRF and SCRF

The thesis only discusses how multiword cache lines and snoopy caches can be implemented on the CRF systems individually. However, it is common for commodity products to have both optimizations. Therefore, it is worth combining the MCRF and the SCRF models and having implementations supporting both optimization.

CRF Translator

The Multiword Base protocol implemented in this project uses its own instruction set, which uses synthetic benchmark to test the correctness. This limit the research possibilities of the protocol. Since Shen has shown that most of the commodity products' memory models can be translated to the CRF [1], it is possible to implement a translator to translate the memory instructions from other instruction set to the CRF instruction. The translator needs to be able to translate the instructions efficiently and accurately in real time. Once the translator is implemented, the effectiveness of different CRF protocols can be compared with other cache coherence protocols.

Appendix A

Definitions of The CRF Model

This chapter presents the original CRF model introduced in Xiaowei Shen's Ph.D. Dissertation [1]. The MCRF and the SCRF are derived from this model.

A.1 CRF Configurations

Figure A-1 defines the CRF's instruction and its system configuration. As can be seen, the CRF instructions include load-local (Loadl), store-local (Storel), commit, reconcile and four types of memory fences. The CRF system contains a memory (MEM) and a set of sites (SITEs). Each site (SITE) has a semantic sache (SACHE), a processor (PROC), processor-to-memory buffer (PMB) and memory-to-processor buffer (MPB). The two buffers are used to buffer the communication message delivered between the SACHE and the PROC. Each SACHE contains a set of data cells, which backups the data recently accessed by the PROC.

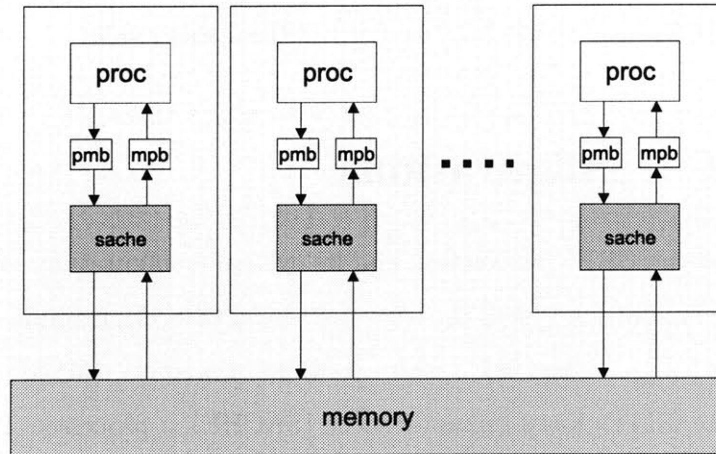
A.2 CRF Rules

The CRF has 2 sets of rules: The first set defines the semantics of Loadl, Storel, Commit and Reconcile instructions. It also includes background rules that govern the data propagation between the semantic caches and the memory. The first set is also known as the Commit-Reconcile (CR) model, because this set of rules itself defines a memory model which is the same as the CRF model but with instructions executed strictly in order; the second set of rules defines the semantics of the instruction reorderings and the memory fences. In the

CRF Instructions

INST	$\equiv$	Loadl(a) [] Storel(a,v)
	[]	Commit(a) [] Reconcile(a)
	[]	Fence$_{rr}$($a1$, $a2$) [] Fence$_{rw}$($a1$, $a2$)
	[]	Fence$_{wr}$($a1$, $a2$) [] Fence$_{ww}$($a1$, $a2$)

CRF System Configurations



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 A-1: Instructions and System Configurations of CRF

remaining of the section, I first present the definitions of the CR model and then present the definitions of the second set of the rules.

A.2.1 CR Model

Loadl and Storel Rules: A Loadl or Storel can be completed if the address is cached in the sache. A Loadl returns the value of the address to the processor and a Storel updates the value of the address in the sache.

CRF-Loadl Rule

Site(sache, $\langle t$,Loadl(a)$\rangle$;pmb, mpb, proc)   if   Cell(a, v, -) $\in$ sache

$\rightarrow$ Site(sache, pmb, mpb|$\langle t$,$v_m\rangle$, proc)

CRF-Storel Rule

Site(Cell(a,-,-) | sache, $\langle t$, Storel(a, v)$\rangle$;pmb, mpb, proc)

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

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

CRF-Commit Rule

Site(sache, $\langle t$,Commit(a)$\rangle$;pmb, mpb, proc)   if   Cell(a,-,Dirty) $\notin$ sache

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

CRF-Reconcile Rule

Site(sache, $\langle t$,Reconcile(a)$\rangle$;pmb, mpb, proc)   if   Cell(a,v,Clean) $\notin$ sache

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

Cache, Writeback and Purge Rules: A sache can obtain a Clean copy of an address from the memory provided that the address is not cached in the sache. A Dirty copy of an address can be written back to the memory, after which the cache state of that address becomes Clean. A Clean copy of an address can be purged from the sache at any time.

$I_2 \Rightarrow$ $I_1 \Downarrow$	Loadl (a')	Storel (a',v')	Fence$_{rr}$ (a'_1,a'_2)	Fence$_{rw}$ (a'_1,a'_2)	Fence$_{wr}$ (a'_1,a'_2)	Fence$_{ww}$ (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$_{rr}$(a_1,a_2)	true	true	true	true	true	true	true	$a_2 \neq a'$
Fence$_{rw}$(a_1,a_2)	true	$a_2 \neq a'$	true	true	true	true	true	true
Fence$_{wr}$(a_1,a_2)	true	true	true	true	true	true	true	$a_2 \neq a'$
Fence$_{ww}$(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 A-2: Instruction Reordering Table

CRF-Cache Rule

Sys(mem, Site(sache, pmb, mpb, proc) | sites) if $a \notin$ sache

$\rightarrow$ Sys(mem, Site(Cell(a,mem[a],Clean) | sache, pmb, mpb, proc) | sites)

CRF-Writeback Rule

Sys(mem, Site(Cell(a,v,Dirty) | sache, pmb, mpb, proc) | sites)

$\rightarrow$ Sys(mem[$a := v$], Site(Cell(a,v,Clean) | sache, pmb, mpb, proc) | sites)

CRF-Purge Rule

Site(Cell(a,-,Clean) | sache, pmb, mpb, proc)

$\rightarrow$ Site(sache, pmb, mpb, proc)

A.3 Rules for Instruction Reorderings and Memory Fences

Figure A-2 summarizes the conditions that two instructions can be reordered (Instruction I_1 followed by instruction I_2 can only be reordered if their corresponding entry in the table is evaluated "true"). It is trivial to derive the rules of instruction reorderings from this table: For each entry in the table that returns true, add a rule to allow the two instructions to be reordered in the processor-to-memory buffer (PMB).

Bibliography

[1] X. Shen. Design and Verification of Adaptive Cache Coherence Protocols. Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, February 2000.

[2] J. Hoe, Arvind. Hardware Synthesis from Term Rewriting Systems. In *Proceedings of the IFIP TC10/WG10.5 Tenth International Conference on Very Large Scale Integration: Systems on a Chip*, pages 595-619, December 1999

[3] J. Goodman. Using Cache Memory To Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124-131, 1983.

[4] J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach, Third Edition. Morgan Kaufman, Chapter 5, pages 100-200, 2002.

[5] B. Kim. An Efficient Buffer Management Method for Cachet. Master's Thesis, Massachusetts Institute of Technology, Cambridge, September 2004.

[6] M. Martin, M. Hill, D. Wood. Token Coherence: Decoupling Performance and Correcntess In *Proceedings of the 30th Annual Internation Symposium on Computer Architecture*, page 182, 2003.

[7] N. Dave. Designing a Reorder Buffer in Bluespec In *Proceedings of the 2nd ACM-IEEE International Conference on Formal Methods and Models for Codesign*, 2004.

[8] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmen, A. kyker and P. Roussel. The Microarchitecture of the Pentium 4 Processor. http://developer.intel.com/technology/itj/q12001/articles/art_2.htm, 2001.

[9] J. Tendler, S. Dodson, S. Fields, H. Le, B. Sinharoy. IBM eserver POWER4 System Microarchitecture. http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.pdf, 2001.

[10] C. Kim, D. Burger and S. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operation Systems*, October 2002.

[11] C. Cowell, C. Moritz and W. Burleson. Improved Modeling and Data Migration for Dynamic Non-Uniform Cache Accesses. In *Proceedings of the 2nd Workshop on Duplicating, Deconstructing, and Debunking*, 2003.

[12] P. Yiannacouras and J. Rose. A Parameterized Automatic Cache Generator for FPGAs. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*, pages 324-327 , December 2003.

 3896 72