An Efficient Buffer Management Method for Cachet

by

Byungsub Kim

B.S., Electrical Engineering
Pohang University of Science and Technology, 2000

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004

Author .....
Department of Electrical Engineering and Computer Science
August 6, 2004

Certified by .....
Arvind
Johnson Professor
Thesis Supervisor

Accepted by .....
Ain A. Sonin
Chairman, Department Committee on Graduate Students

An Efficient Buffer Management Method for Cachet

by

Byungsub Kim

B.S., Electrical Engineering

Pohang University of Science and Technology, 2000

Submitted to the Department of Electrical Engineering and Computer Science
on August 6, 2004, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

We propose an efficient buffer management method for Cachet [7], called BCachet. Cachet is an adaptive cache coherence protocol based on a mechanism-oriented memory model called Commit-Reconcile & Fences (CRF) [1]. Although Cachet is theoretically proved to be sound and live, a direct implementation of Cachet is not feasible because it requires too expensive hardware.

We greatly reduced the hardware cost for buffer management in BCachet without changing the memory model and the adaptive nature of Cachet. Hardware cost for the incoming message buffer of the memory site is greatly reduced from $P \times N$ FIFOs to two FIFOs in BCachet where P is the number of sites and N is the number of address lines in a memory unit. We also reduced the minimum size of suspended message buffer per memory site from $(\log_2 P + V) \times P \times rq_{max}$ to $\log_2 P$ where V is the size of a memory block in terms of bits and rq_{max} is the maximum number of request messages per cache.

BCachet has three architectural merits. First, BCachet separates buffer management units for deadlock avoidance and those units for livelock avoidance so that a designer has an option in the liveness level and the corresponding hardware cost: (1) allows a livelock with an extremely low probability and saves hardware cost for fairness control. (2) does not allow a livelock at all and accept hardware cost for fairness control. Second, a designer can easily parameterize the sizes of buffer units to explore the cost-performance curves without affecting the soundness and the liveness. Because usual sizes of buffer management units are much larger than the minimum sizes of those units that guarantee the liveness and soundness of the system, a designer can easily find optimum trade-off point for those units by changing size parameters and running simulation. Third, since BCachet is almost linear under the assumption of a reasonable number of sites, BCachet is very scalable. Therefore, it can be used to for very large scale multiprocessor systems.

Thesis Supervisor: Arvind
Title: Johnson Professor

Acknowledgments

I would like to thank Arvind, my adviser for his guidance and support. He showed me lots of interesting issues which I had not recognized before. Sometimes, he surprised me with his endless enthusiasm on research by staying very late in his office or coming to the office on holiday. While I was working on this thesis, he led me toward the right way with his deep and broad knowledge and smart logic when I was on the wrong track. I could not have finished this thesis without his kind advice.

I also want to say thanks to all of my colleagues at Computation Structures Group at MIT Computer Science and Artificial Intelligence Laboratory. I want to thank Man Cheuk Ng, my funny office mate. When I became "Crazy Man," he willingly became "Cool Man." We spent lots of time in arguing about research as well as in playing basketball, chatting, games, joking, and hunting free foods. I want to mention Vinson Lee, " the only normal guy," in "my thanks list" not only because he mentioned me in the acknowledgment of his thesis but also he gave me my nick name, "Crazy Man." He was one of the funniest guys I ever met at MIT. I also want to say thanks to "CSG Axes of Evil," Edward Gookwon Suh, Jaewook Lee, and Daniel L. Rosendband. Actually, they are not as evil as their nick name means. I would like to thank to Karen Brennan, Nirav Dave, and Winnie Cheng, we had a great time, hanging around, playing card, and playing squash. Jacob Schwartz taught me a lot about Bluespec. Charles O'Donnell, a "Yankee fan" often informed me about the standing of the Yankees and Red Sox in semi-real time. Daihyun Lim, my 6.374 project mate, lab mate, roommate helped me a lot in my two years at MIT. I want to say thanks to Prabhat Jain for his advice about life at MIT.

I also want to say thanks to other friends. My other 6.374 project friend, Alex F Narvaez brought me to some very nice places and had a great time. Joungkeun Lim helped me to pursue an exercise project, called "Mom Zang Project." Jason Kim told and advised me a lot about MIT based on his experience at MIT. His wisdom about life at MIT helped me out when I was jammed. I would like to thank all the other people who helped me. Not mentioning those names here does not mean I forgot about their help.

I cannot say thanks enough to my mother. Her love and support helped me not only to pursue my study here but also to overcome hard times in my life. She always takes care of me, wherever I am, whatever I do. When I was sick, she was sitting beside me. When I got discouraged, she encouraged me. When I gave up, she did not give up. Without her strong love, I could not finish this study. This work belongs to her strong love to her child.

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

Contents

Chapter 1

Introduction

In a Distributed Shared Memory (DSM) system processors, and memory are distributed and communicate with each other through a network. DSM system have long communication latencies and usually employ hardware and software caching technique to overcome this latency. DSM systems have provided a fertile ground for research in cache coherence protocols [2, 3, 11, 18, 23, 20, 6, 17, 19]. There is an emerging consensus that a single fixed protocol is unlikely to provide satisfactory performance for a range of protocols [21, 13, 22, 9, 15]. Consequently, there has been a great interest in adaptive protocols that can monitor the behavior of a program and dynamically adjust to improve its performance. There are few examples of adaptive protocols in use - both their performance and correctness have been incredibly difficult to establish.

This thesis is about further development of an adaptive cache coherence protocol known as Cachet, which was designed by Xiaowei Shen in late nineties [5, 7]. Cachet has several attractive attributes:

1. Cachet is composed of three micro-protocols, Base, Writer-Push, and Migratory. Each micro-protocol is optimized for different access patterns, so that a system can achieve performance improvement by adaptively changing from one micro-protocol to another based on program execution monitoring and some adaptivity policy.

2. Cachet provides two sets of rules - mandatory and voluntary, and guarantees that the use of voluntary rules cannot affect the correctness of the protocol. Cachet has solved a major problem in adaptive protocol verification by such a separation of rules. Although a voluntary action may be done based on a wrong prediction, the system

does not have to recover from the voluntary action because firing voluntary rule is always safe in Cachet.

3. Cachet supports the Commit-Reconcile and Fences (CRF) [1] memory model. CRF is a low-level mechanism-oriented memory model for architects and compiler writers. It is universal in the sense that almost all other memory models can be described using CRF primitives via translation [1]. Consequently, a system that uses Cachet can support a variety of memory models by one protocol.

In spite of all these wonderful properties, Cachet needs more development before it can be used in a practical system. Cachet provides no policy guidelines about adaptivity. Little is known about what can or should be monitored about a program and used to provide dynamic feedback to the cache system. The other major issue in the implementation of Cachet in a DSM system is the management of communication buffers. Xiaowei Shen has chosen to describe the Cachet protocol in an abstract manner where the clarity and correctness issues have dominated the buffer management issues.

For a system with N address lines and P processors in which each processor can generate rq_{max} writeback messages at a time, a direct implementation of Cachet requires $N \times P$ FIFO buffers at the input of memory. For even a moderate size system this number is too large and thus, practically infeasible. Reordering of messages can reduce buffer sizes at the expense of making buffer management more difficult. Cachet also requires storage to hold up to P suspended writeback messages per address line if each address line has its own storage, or Cachet requires storage to hold up to $P \times rq_{max}$ suspended writeback messages per memory unit if the storage is shared by all address lines.

The main contribution of this thesis is to provide BCachet, a modified Cachet protocol, with practical buffer management characteristics. BCachet requires only two FIFOs at the input of memory. BCachet also reduces the need for suspended message buffers to one processor name tag per address line if each address line has its own suspended message buffers, or one processor name tag per memory unit if the suspended buffer is shared by all address lines. The reduced buffer requirement does not cause BCachet to be less flexible or adaptive than Cachet.

Thesis organization: We give a brief overview of CRF and Cachet in Chapter 2. In Chapter 3, we define HWb as an intermediate step toward describing BCachet in Chapter 4.

In Chapter 5, we briefly show the correctness of BCachet, and finally in Chapter 6, we offer our conclusions and some ideas for future work.

Chapter 2

Cachet

Cachet [7] is a protocol for the mechanism-oriented memory model called Commit-Reconcile & Fences (CRF) [1]. We give overviews of CRF and Cachet in Sections 2.1 and Section 2.2, respectively. These descriptions are taken almost verbatim from papers on CRF and Cachet [1, 7] and Xiaowei Shen's doctoral dissertation [7].

2.1 Commit Reconcile & Fences (CRF)

A system diagram of CRF memory model is shown in Figure 2-1. It includes a global memory (MEM), processors (PROC), memory to processor buffer (MPB), processor to memory buffer (PMB), and semantic caches called *sache*.

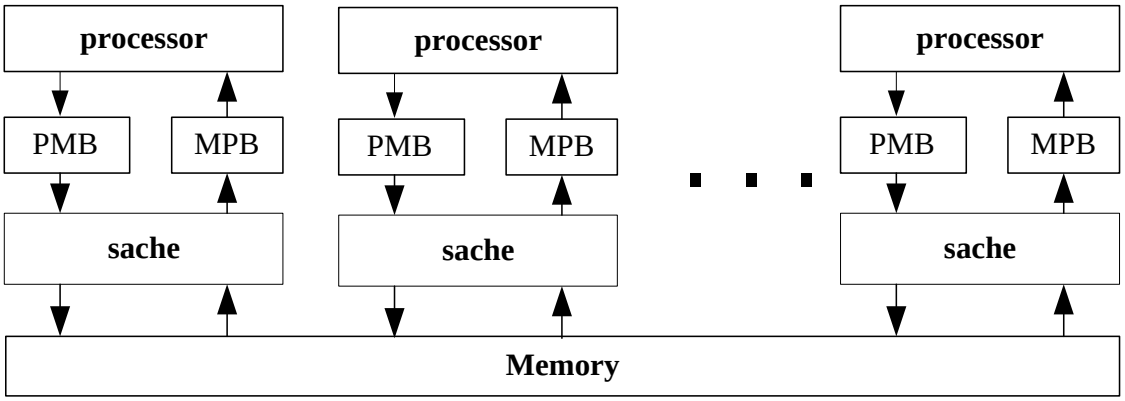

Figure 2-1: CRF System Overview

We use the syntax shown in Figure 2-2 to describe the detailed structure of the CRF system. Notice every request from a processor to memory contains a tag t, which the mem-

ory system uses in responding to a particular request. Messages in MPB can be reordered arbitrarily because of these tags. Messages in PMB can also be reordered but are subject to some restrictions as will be discussed shortly.

SYS	≡	Sys(MEM,SITEs)	*System*
SITEs	≡	SITE [] SITE|SITEs	*Set of Sites*
SITE	≡	Site(SACHE,PMB,MPB,PROC)	*Site*
SACHE	≡	ϵ [] Cell(a,v,CSTATE)|SACHE	*Semantic Cache*
CSTATE	≡	Clean [] Dirty	*Cache State*
PMB	≡	< t,INST>;PMB	*Processor-to-Memory Buffer*
MPB	≡	< t,REPLY>|MPB	*Memory-to-Processor Buffer*
REPLY	≡	v [] Ack	*Reply*
INST	≡	Loadl(a) [] Storel(a,v) [] Commit(a) [] Reconcile(a) [] Fence$_{rr}$(a_1,a_2) [] Fence$_{rw}$(a_1,a_2) [] Fence$_{wr}$(a_1,a_2) [] Fence$_{ww}$(a_1,a_2)	*Instruction*

Figure 2-2: TRS expression of CRF

Two cache states, Clean and Dirty, are used in CRF. CRF defines execution of memory instructions in terms of the cache state change and data movement between saches and the memory.

CRF has eight memory instructions (please see Table 2.1). Conventional Load and Store instructions are split into two local memory instructions and two global instructions to achieve finer-grained memory operation. The local instructions, Loadl and Storel, can be performed if the address is cached in the sache. If the address is in the sache, Loadl reads the value and Storel writes a value to the sache block. Two global memory instructions, Commit and Reconcile perform a synchronization of sache to the memory. Commit guarantees that there is no Dirty copy of the address in the sache after it is completed while Reconcile guarantees that there is no Clean copy in the sache after it is complete.

Four fence instructions are used to control the reordering of memory instructions in PMB. Each fence operation has two arguments, pre-address and post-address. A fence instruction does not allow the reordering between two instructions of the addresses in certain situations. The instruction reordering table that defines the fence operation is shown in Table 2.1.

The interactions between memory and sache is defined as background rules. According to *CRF-Cache* rule, a sache can obtain a Clean copy if the address is not cached. According to *CRF-Writeback* rule, a Dirty copy can be written back to the memory and become a Clean copy. A Clean copy can be purged by *CRF-Purge* rule.

2.2 Cachet

The Cachet system models a distributed shared memory system as shown in Figure 2-3. In this model, message buffer are used to model a network delay. Processor, processor-to-memory, and memory-to-processor buffer are the same as in the CRF model. Cache elements in Cachet have more possible states than simply *Clean* and *Dirty*. Also unlike CRF, in Cachet memory cells have several states associated with them. Message queues of caches and memory's are modeled by point-to-point buffers where two messages can be reordered if addresses, destinations, or sources in the two messages are different (see Figure 2-4).

Before giving the details of Cachet, we explain the three micro-protocols that are used to compose it.

2.2.1 Micro-Protocols

Cachet employs three different micro-protocols, Base, Writer-Push, and Migratory. Characteristic of each micro-protocol is described as follows.

Processor Rules

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

Background Rules

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

Instruction Reordering

I_1 \ I_2	$Loadl$ (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')
$Loadl(a)$	$true$	$a \neq a'$	$a \neq a'_1$	$a \neq a'_1$	$true$	$true$	$true$	$true$
$Storel(a, v)$	$a \neq a'$	$a \neq a'$	$true$	$true$	$true$	$true$	$a \neq a'$	$true$
$Fence_{rr}(a_1, a_2)$	$true$	$true$	$true$	$true$	$true$	$true$	$true$	$a_2 \neq a'$
$Fence_{rw}(a_1, a_2)$	$true$	$a_2 \neq a'$	$true$	$true$	$true$	$true$	$true$	$true$
$Fence_{wr}(a_1, a_2)$	$true$	$true$	$true$	$true$	$true$	$true$	$true$	$a_2 \neq a'$
$Fence_{ww}(a_1, a_2)$	$true$	$a_2 \neq a'$	$true$	$true$	$true$	$true$	$true$	$true$
$Commit(a)$	$true$	$true$	$true$	$true$	$a \neq a'_1$	$a \neq a'_1$	$true$	$true$
$Reconcile(a)$	$a \neq a'$	$true$	$true$	$true$	$true$	$true$	$true$	$true$

Table 2.1: Summary of CRF Rules

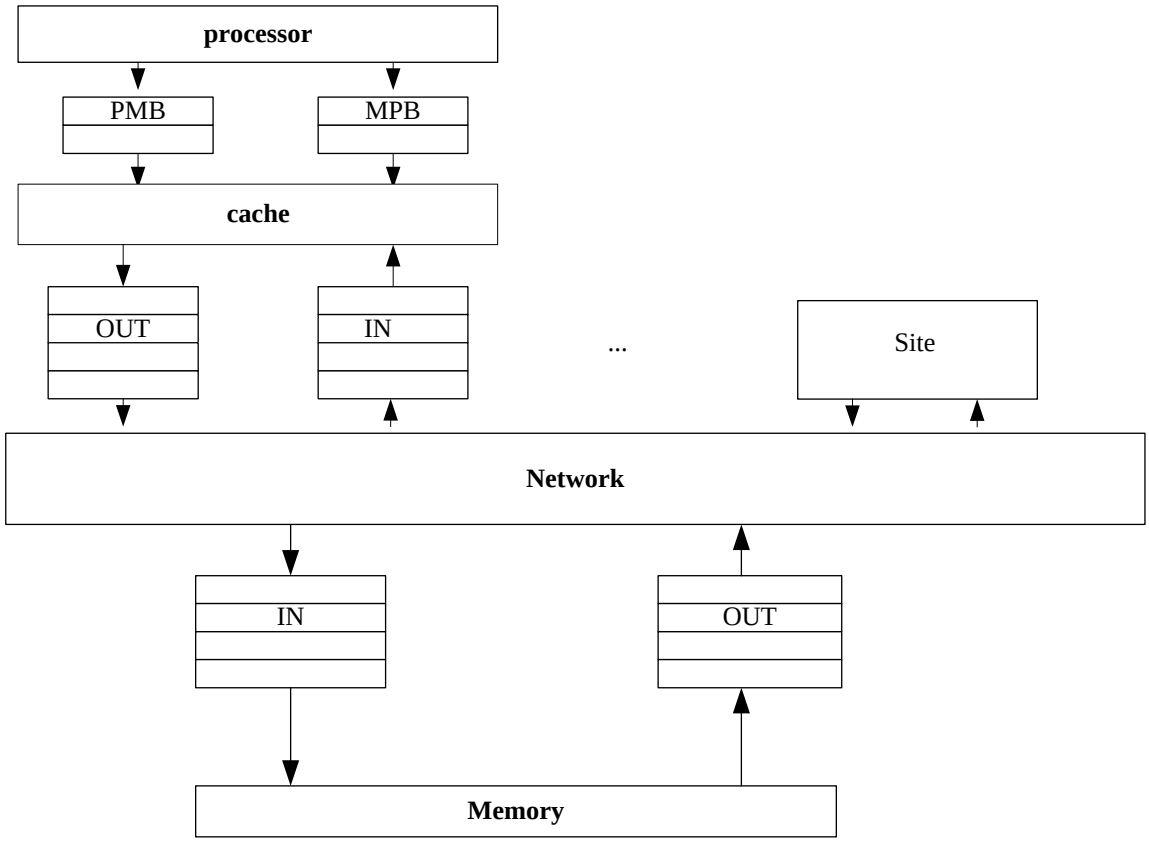

Figure 2-3: Cachet System Overview

$$msg_1 \odot msg_2 \equiv msg_2 \odot msg_1$$
$$if\ Dest(msg_1) \neq Dest(msg_2) \vee Addr(msg_1) \neq Addr(msg_2) \vee Src(msg_1) \neq Src(msg_2)$$

Figure 2-4: Point to Point Message Passing

- Base: Base protocol is the most straightforward implementation of CRF. It uses the memory as a rendezvous point for reading and writing, that is, to read or write a value globally, a cache must access the memory. In this protocol, a cache must write back a Dirty copy on Commit and purge a Clean copy on Reconcile to guarantee that the memory stores the most recently updated global value.

- Writer-Push: Writer-Push is optimized for the case where load operations are more frequent than store operations. It uses caches as rendezvous points for reading and the memory as a rendezvous point for writing, that is, a cache must access the memory to write a value globally. In this protocol, a cache must write back a Dirty copy on Commit and the memory is responsible to purge all other copies before update the memory value to guarantee that the memory and the Writer-Push cache blocks store the most recently updated global value.

- Migratory: Migratory is optimized for the case where an address is exclusively owned by a processor for a long time. It uses the cache as a rendezvous point for reading and writing, that is, a cache does not need to access to the memory for global reading and writing operation. In this protocol, to guarantee that the Migratory cache block stores the most recently updated global value, the memory is responsible for purging the Migratory cache block and retrieving the most recently updated global value on requests from other sites.

The difference of Base, Writer-Push, and Migratory are shown in Table 2.2.

Micro-Protocol	Commit on Dirty	Reconcile on Clean	Cache Miss
Base	update memory	purge local clean copy	retrieve data from memory
Writer-Push	purge all clean copy update memory		retrieve data from memory
Migratory			flush exclusive copy update memory retrieve data from memory

Table 2.2: Different Treatment of Commit, Reconcile, and Cache Miss

2.2.2 Cache and Memory States, and Message Types in Cachet

Figure 2-5 shows various elements of Cachet in our syntax. It is followed by the explanations of each cache and memory state, and message type

SYS	≡	Sys(MSITE,SITEs)	*System*
MSITE	≡	Msite(MEM,IN,OUT)	*Memory Site*
MEM	≡	ϵ [] Cell(a,v,MSTATE)|MEM	*Memory*
SITEs	≡	SITE [] SITE|SITEs	*Set of Cache Sites*
SITE	≡	Site(id,CACHE,IN,OUT,PMB,MPB,PROC)	*Cache Site*
IN	≡	ϵ [] MSG$\odot$IN	*Incoming Queue*
OUT	≡	ϵ [] MSG$\odot$OUT	*Outgoing Queue*
MSG	≡	Msg(src,$dest$,CMD,a,v)	*Message*
MSTATE	≡	Cw[DIR] [] Tw[DIR,SM] [] Cm[id] [] Tm[id,SM] [] T'm[id]	*Memory State*
DIR	≡	ϵ [] id|DIR	*Directory*
SM	≡	ϵ [] (id,v)|SM	*Suspended Message*
CSTATE	≡	Clean_b [] Dirty_b [] Clean_w [] Dirty_w [] Clean_m [] Dirty_m [] WbPending [] CachePending	*Cache State*
CMD	≡	CacheReq [] Wb [] Down_{wb} [] Down_{mw} [] DownV_{mw} [] Down_{mb} [] DownV_{mb} [] Cache_b [] Cache_w [] Up_{wm} [] WbAck_b [] DownReq_{wb} [] DownReq_{mw} [] DownReq_{mb}	*Command*

Figure 2-5: TRS Expression of Cachet

Cache States of Cachet

Cachet protocol employs six stable cache states and two transient states. These states are list as follows.

- $Clean_b$: Clean state of Base.
- $Dirty_b$: Dirty state of Base.
- $Clean_w$: Clean state of Writer-Push.
- $Dirty_w$: Dirty state of Writer-Push.
- $Clean_m$: Clean state of Migratory.
- $Dirty_m$: Dirty state of Migratory.
- $WbPending$: The transient state that indicate a dirty copy of the address is being written back to the memory.

- *CachePending*: The transient state that indicate a data copy of the address is being retrieved from the memory.
- *Invalid*: The address is not cached.

Memory States of Cachet

Cachet protocol employs two stable memory states and three transient states. Two transient states contain suspended writeback messages.

- Cw[*dir*]: The address is cached under Writer-Push in the sites whose identifiers are recorded in *dir*.
- Cm[*id*]: The address is cached under Migratory in the site whose identifier is *id*.
- Tw[*dir*,*sm*]: The address is cached under Writer-Push in the sites whose identifiers are recorded in *dir*. The suspended message buffer *sm* stores suspended writeback messages. The memory has sent $DownReq_{wb}$ to the sites whose identifiers are recorded in *dir*.
- Tm[*id*,*sm*]: The address is cached under Migratory in the site whose identifier is *id*. The suspended message buffer *sm* stores suspended writeback messages. The memory has sent a $DownReq_{mb}$ to the site whose identifier is *id* or it has sent a $DownReq_{mw}$ followed by a $DownReq_{wb}$ to the site.
- T'm[*id*]: The address is cached under Migratory in the site whose identifier is *id*. The memory has sent a $DownReq_{mw}$ to the site whose identifier is *id*.

Messages of Cachet

To achieve high adaptivity and seamless integration, Cachet employs eighteen messages: ten memory-to-cache messages and eight cache-to-home messages. Based on behavioral similarity, we classify them into eight categories as follows.

- ***CacheReq*** requests a memory to send a data copy.
  - *CacheReq* informs that the cache request a data copy from the memory.
- ***Cache*** carries a data copy to the cache.

- $Cache_b$ carries a Base copy to the cache.
- $Cache_w$ carries a Writer-Push copy to the cache.
- $Cache_m$ carries a Migratory copy to the cache.

- ***Up*** informs its target cache that a memory intends to upgrade the micro-protocol of the block.
  - Up_{wm} informs the cache that the memory intends to upgrade a cache block from Writer-Push to Migratory.

- ***Wb*** carries a value to be written back to the memory.
  - Wb_b carries a data copy of dirty Base block, which will be written back to the memory.
  - Wb_w carries a data copy of dirty Writer-Push block, which will be written back to the memory.

- ***WbAck*** informs its target cache of the completion of a writing back operation in memory.
  - $WbAck_b$ informs the cache that writing back of the copy has been done and allows the cache to retain a Base copy.
  - $WbAck_w$ informs the cache that writing back of the copy has been done and allows the cache to retain a Writer-Push copy.
  - $WbAck_m$ informs the cache that writing back of the copy has been done and allows the cache to retain a Migratory copy.

- ***DownReq*** requests a cache to downgrade the micro-protocol of the block.
  - $DownReq_{wb}$ informs the cache of the request from the memory for downgrading from Writer-Push to Base.
  - $DownReq_{mw}$ informs the cache of the request from the memory for downgrading from Migratory to Writer-Push.
  - $DownReq_{mb}$ informs the cache of the request from the memory for downgrading from Migratory to Base.

- ***Down*** informs its target memory that the cache has downgraded the micro-protocol.
  - $Down_{wb}$ informs the memory that the cache block has downgraded from Writer-Push to Base.
  - $Down_{mw}$ informs the memory that the cache block has downgraded from Migratory to Writer-Push.
  - $Down_{mb}$ informs the memory that the cache block has downgraded from Migratory to Base.
- ***DownV*** informs its target memory that the cache has downgraded and carries the value to be written back.
  - $DownV_{mb}$ informs its target memory that the cache has downgraded from Migratory to Base and carries a data copy to be written back.
  - $DownV_{mw}$ informs its target memory that the cache has downgraded from Migratory to Writer-Push and carries a data copy to be written back.

2.2.3 Specification of Cachet

Table 2.3, Table 2.4, and Table 2.5 give all the rules of Cachet. Detail information about Cachet, including the proof of its soundness and liveness can be found in Xiaowei Shen's thesis [7].

Table 2.3: Cachet: The Processor Rules

Mandatory Processor Rules				
Instruction	**Cstate**	**Action**	**Next Cstate**	
$Loadl(a)$	$Cell(a, v, Clean_b)$	*retire*	$Cell(a, v, Clean_b)$	P1
	$Cell(a, v, Dirty_b)$	*retire*	$Cell(a, v, Dirty_b)$	P2
	$Cell(a, v, Clean_w)$	*retire*	$Cell(a, v, Clean_w)$	P3
	$Cell(a, v, Dirty_w)$	*retire*	$Cell(a, v, Dirty_w)$	P4
	$Cell(a, v, Clean_m)$	*retire*	$Cell(a, v, Clean_m)$	P5
	$Cell(a, v, Dirty_m)$	*retire*	$Cell(a, v, Dirty_m)$	P6
	$Cell(a, v, WbPending)$	*stall*	$Cell(a, v, WbPending)$	P7
	$Cell(a, -, CachePending)$	*stall*	$Cell(a, -, CachePending)$	P8
	$a \notin cache$	$stall, < CacheReq, a > \rightarrow H$	$Cell(a, -, CachePending)$	P9
$Storel(a, v)$	$Cell(a, v, Clean_b)$	*retire*	$Cell(a, v, Dirty_b)$	P10
	$Cell(a, v, Dirty_b)$	*retire*	$Cell(a, v, Dirty_b)$	P11
	$Cell(a, v, Clean_w)$	*retire*	$Cell(a, v, Dirty_w)$	P12

Continued on next page

Table 2.3 – continued from previous page

Instruction	Cstate	Action	Next Cstate	
	$Cell(a, v, Dirty_w)$	*retire*	$Cell(a, v, Dirty_w)$	P13
	$Cell(a, v, Clean_m)$	*retire*	$Cell(a, v, Dirty_m)$	P14
	$Cell(a, v, Dirty_m)$	*retire*	$Cell(a, v, Dirty_m)$	P15
	$Cell(a, v_1, WbPending)$	*stall*	$Cell(a, v_1, WbPending)$	P16
	$Cell(a, -, CachePending)$	*stall*	$Cell(a, -, CachePending)$	P17
	$a \notin cache$	$stall, < CacheReq, a > \rightarrow H$	$Cell(a, -, CachePending)$	P18
$Commit(a)$	$Cell(a, v, Clean_b)$	*retire*	$Cell(a, v, Clean_b)$	P19
	$Cell(a, v, Dirty_b)$	$stall, < Wb_b, a, v > \rightarrow H$	$Cell(a, v, WbPending)$	P20
	$Cell(a, v, Clean_w)$	*retire*	$Cell(a, v, Clean_w)$	P21
	$Cell(a, v, Dirty_w)$	$stall, < Wb_w, a, v > \rightarrow H$	$Cell(a, v, WbPending)$	P22
	$Cell(a, v, Clean_m)$	*retire*	$Cell(a, v, Clean_m)$	P23
	$Cell(a, v, Dirty_m)$	*retire*	$Cell(a, v, Dirty_m)$	P24
	$Cell(a, v, WbPending)$	*stall*	$Cell(a, v, WbPending)$	P25
	$Cell(a, -, CachePending)$	*stall*	$Cell(a, -, CachePending)$	P26
	$a \notin cache$	*retire*	$a \notin cache$	P27
$Reconcile(a)$	$Cell(a, -, Clean_b)$	*retire*	$a \notin cache$	P28
	$Cell(a, v, Dirty_b)$	*retire*	$Cell(a, v, Dirty_b)$	P29
	$Cell(a, v, Clean_w)$	*retire*	$Cell(a, v, Clean_w)$	P30
	$Cell(a, v, Dirty_w)$	*retire*	$Cell(a, v, Dirty_w)$	P31
	$Cell(a, v, Clean_m)$	*retire*	$Cell(a, v, Clean_m)$	P32
	$Cell(a, v, Dirty_m)$	*retire*	$Cell(a, v, Dirty_m)$	P33
	$Cell(a, v, WbPending)$	*stall*	$Cell(a, v, WbPending)$	P34
	$Cell(a, -, CachePending)$	*stall*	$Cell(a, -, CachePending)$	P35
	$a \notin cache$	*retire*	$a \notin cache$	P36

Table 2.4: Cachet: The Cache Engine Rules

Voluntary C-engine Rules				
Msg form H	**Cstate**	**Action**	**Next Cstate**	
	$Cell(a, -, Clean_b)$		$a \notin cache$	C1
	$Cell(a, v, Dirty_b)$	$< Wb_b, a, v > \rightarrow H$	$Cell(a, v, WbPending)$	C2
	$Cell(a, v, Clean_w)$	$< Down_{wb}, a > \rightarrow H$	$Cell(a, v, Clean_b)$	C3
	$Cell(a, v, Dirty_w)$	$< Down_{wb}, a > \rightarrow H$	$Cell(a, v, Dirty_b)$	C4
		$< Wb_w, a, v > \rightarrow H$	$Cell(a, v, WbPending)$	C5
	$Cell(a, v, Clean_m)$	$< Down_{mw}, a > \rightarrow H$	$Cell(a, v, Clean_w)$	C6
		$< Down_{mb}, a > \rightarrow H$	$Cell(a, v, Clean_b)$	C7
	$Cell(a, v, Dirty_m)$	$< DownV_{mw}, a, v > \rightarrow H$	$Cell(a, v, Clean_w)$	C8
		$< DownV_{mb}, a, v > \rightarrow H$	$Cell(a, v, Clean_b)$	C9
	$a \notin cache$	$< CacheReq, a > \rightarrow H$	$Cell(a, -, CachePending)$	C10

Mandatory C-engine Rules
Continued on next page

Table 2.4 – continued from previous page

Msg form H	Cstate	Action	Next Cstate	
$< Cache_b, a, v >$	$Cell(a, -, CachePending)$		$Cell(a, v, Clean_b)$	C11
$< Cache_w, a, v >$	$Cell(a, -, Clean_b)$		$Cell(a, v, Clean_w)$	C12
	$Cell(a, v_1, Dirty_b)$		$Cell(a, v_1, Dirty_w)$	C13
	$Cell(a, v_1, WbPending)$		$Cell(a, v_1, WbPending)$	C14
	$Cell(a, -, CachePending)$		$Cell(a, v, Clean_w)$	C15
	$a \notin cache$		$Cell(a, v, Clean_w)$	C16
$< Cache_m, a, v >$	$Cell(a, -, Clean_b)$		$Cell(a, v, Clean_m)$	C17
	$Cell(a, v_1, Dirty_b)$		$Cell(a, v_1, Dirty_m)$	C18
	$Cell(a, v_1, WbPending)$		$Cell(a, v_1, WbPending)$	C19
	$Cell(a, -, CachePending)$		$Cell(a, v, Clean_m)$	C20
	$a \notin cache$		$Cell(a, v, Clean_m)$	C21
$< Up_{wm}, a >$	$Cell(a, v, Clean_b)$		$Cell(a, v, Clean_b)$	C22
	$Cell(a, v, Dirty_b)$		$Cell(a, v, Dirty_b)$	C23
	$Cell(a, v, Clean_w)$		$Cell(a, v, Clean_m)$	C24
	$Cell(a, v, Dirty_w)$		$Cell(a, v, Dirty_m)$	C25
	$Cell(a, v, WbPending)$		$Cell(a, v, WbPending)$	C26
	$Cell(a, -, CachePending)$		$Cell(a, -, CachePending)$	C27
	$a \notin cache$		$a \notin cache$	C28
$< WbAck_b, a >$	$Cell(a, v, WbPending)$		$Cell(a, v, Clean_b)$	C29
$< WbAck_w, a >$	$Cell(a, v, WbPending)$		$Cell(a, v, Clean_w)$	C30
$< WbAck_m, a >$	$Cell(a, v, WbPending)$		$Cell(a, v, Clean_m)$	C31
$< DownReq_{wb}, a >$	$Cell(a, v, Clean_b)$		$Cell(a, v, Clean_b)$	C32
	$Cell(a, v, Dirty_b)$		$Cell(a, v, Dirty_b)$	C33
	$Cell(a, v, Clean_w)$	$< Down_{wb}, a > \rightarrow H$	$Cell(a, v, Clean_b)$	C34
	$Cell(a, v, Dirty_w)$	$< Down_{wb}, a > \rightarrow H$	$Cell(a, v, Dirty_b)$	C35
	$Cell(a, v, WbPending)$		$Cell(a, v, WbPending)$	C36
	$Cell(a, -, CachePending)$		$Cell(a, -, CachePending)$	C37
	$a \notin cache$		$a \notin cache$	C38
$< DownReq_{mw}, a >$	$Cell(a, v, Clean_b)$		$Cell(a, v, Clean_b)$	C39
	$Cell(a, v, Dirty_b)$		$Cell(a, v, Dirty_b)$	C40
	$Cell(a, v, Clean_w)$		$Cell(a, v, Clean_w)$	C41
	$Cell(a, v, Dirty_w)$		$Cell(a, v, Dirty_w)$	C42
	$Cell(a, v, Clean_m)$	$< Down_{mw}, a > \rightarrow H$	$Cell(a, v, Clean_w)$	C43
	$Cell(a, v, Dirty_m)$	$< DownV_{mw}, a, v > \rightarrow H$	$Cell(a, v, Clean_w)$	C44
	$Cell(a, v, WbPending)$		$Cell(a, v, WbPending)$	C45
	$Cell(a, -, CachePending)$		$Cell(a, -, CachePending)$	C46
	$a \notin cache$		$a \notin cache$	C47
$< DownReq_{mb}, a >$	$Cell(a, v, Clean_b)$		$Cell(a, v, Clean_b)$	C48
	$Cell(a, v, Dirty_b)$		$Cell(a, v, Dirty_b)$	C49
	$Cell(a, v, Clean_w)$	$< Down_{wb}, a > \rightarrow H$	$Cell(a, v, Clean_b)$	C50
	$Cell(a, v, Dirty_w)$	$< Down_{wb}, a > \rightarrow H$	$Cell(a, v, Dirty_b)$	C51
	$Cell(a, v, Clean_m)$	$< Down_{mb}, a > \rightarrow H$	$Cell(a, v, Clean_b)$	C52

Continued on next page

Table 2.4 – continued from previous page

Msg form H	Cstate	Action	Next Cstate	
	$Cell(a, v, Dirty_m)$	$< DownV_{mb}, a, v > \rightarrow H$	$Cell(a, v, Clean_b)$	C53
	$Cell(a, v, WbPending)$		$Cell(a, v, WbPending)$	C54
	$Cell(a, -, CachePending)$		$Cell(a, -, CachePending)$	C55
	$a \notin cache$		$a \notin cache$	C56

Table 2.5: Cachet: The Memory Engine Rules

Voluntary M-engine Rules				
	Mstate	**Action**	**Next Mstate**	
	$Cell(a, v, Cw[dir])(id \notin dir)$	$< Cache_w, a, v > \rightarrow id$	$Cell(a, v, Cw[id\|dir])$	M1
	$Cell(a, v, Cw[id])$	$< Up_{wm}, a > \rightarrow id$	$Cell(a, v, Cm[id])$	M2
	$Cell(a, v, Cw[\epsilon])$	$< Cache_m, a, v > \rightarrow id$	$Cell(a, v, Cm[id])$	M3
	$Cell(a, v, Cw[dir])$	$< DownReq_{wb}, a > \rightarrow dir$	$Cell(a, v, Tw[dir, \epsilon])$	M4
	$Cell(a, v, Cm[id])$	$< DownReq_{mw}, a > \rightarrow id$	$Cell(a, v, T'm[id])$	M5
		$< DownReq_{mb}, a > \rightarrow id$	$Cell(a, v, Tm[id, \epsilon])$	M6
	$Cell(a, v, T'm[id])$	$< DownReq_{wb}, a > \rightarrow id$	$Cell(a, v, Tm[id, \epsilon])$	M7

Mandatory M-engine Rules				
Msg from id	**Mstate**	**Action**	**Next Mstate**	
$< CacheReq, a >$	$Cell(a, v, Cw[dir])(id \notin dir)$	$< Cache_b, a, v > \rightarrow id$	$Cell(a, v, Cw[dir])$	M8
	$Cell(a, v, Tw[dir, sm])(id \notin dir)$	*stall message*	$Cell(a, v, Tw[dir, sm])$	M9
	$Cell(a, v, Cw[dir])(id \in dir)$		$Cell(a, v, Cw[dir])$	M10
	$Cell(a, v, Tw[dir, sm])(id \in dir)$		$Cell(a, v, Tw[dir, sm])$	M11
	$Cell(a, v, Cm[id_1])(id \neq id_1)$	*stall message* $< DownReq_{mw}, a > \rightarrow id_1$	$Cell(a, v, T'm[id_1])$	M12
	$Cell(a, v, T'm[id_1])$ $(id \neq id_1)$	*stall message*	$Cell(a, v, T'm[id_1])$	M13
	$Cell(a, v, Tm[id_1, sm])$ $(id \neq id_1)$	*stall message*	$Cell(a, v, Tm[id_1, sm])$	M14
	$Cell(a, v, Cm[id])$		$Cell(a, v, Cm[id])$	M15
	$Cell(a, v, T'm[id])$		$Cell(a, v, T'm[id])$	M16
	$Cell(a, v, Tm[id, sm])$		$Cell(a, v, Tm[id, sm])$	M17
$< Wb_b, a, v >$	$Cell(a, v_1, Cw[dir])(id \notin dir)$	$< DownReq_{wb}, a > \rightarrow dir$	$Cell(a, v_1, Tw[dir, (id, v)])$	M18
	$Cell(a, v_1, Tw[dir, sm])$ $(id \notin dir)$		$Cell(a, v_1, Tw[dir, sm\|(id, v)])$	M19
	$Cell(a, v_1, Cw[id\|dir])$	$< DownReq_{wb}, a > \rightarrow dir$	$Cell(a, v_1, Tw[dir, (id, v)])$	M20
	$Cell(a, v_1, Tw[id\|dir, sm])$		$Cell(a, v_1, Tw[dir, sm\|(id, v)])$	M21
	$Cell(a, v_1, Cm[id_1])(id \neq id_1)$	$< DownReq_{mb}, a > \rightarrow id_1$	$Cell(a, v_1, Tm[id_1, (id, v)])$	M22
	$Cell(a, v_1, T'm[id_1])$ $(id \neq id_1)$	$< DownReq_{wb}, a > \rightarrow id_1$	$Cell(a, v_1, Tm[id_1, (id, v)])$	M23
	$Cell(a, v_1, Tm[id_1, sm])$		$Cell(a, v_1, Tm[id_1, sm\|(id, v)])$	M24

Continued on next page

Table 2.5 – continued from previous page

Msg from id	Mstate	Action	Next Mstate	
	$(id \neq id_1)$			
	$Cell(a, v_1, Cm[id])$		$Cell(a, v_1, Tw[\epsilon, (id, v)])$	M25
	$Cell(a, v_1, T'm[id])$		$Cell(a, v_1, Tw[\epsilon, (id, v)])$	M26
	$Cell(a, v_1, Tm[id, sm])$		$Cell(a, v_1, Tw[\epsilon, sm\|(id, v)])$	M27
$< Wb_w, a, v >$	$Cell(a, v_1, Cw[dir])(id \notin dir)$	$< DownReq_{wb}, a > \rightarrow dir$	$Cell(a, v_1, Tw[dir, (id, v)])$	M28
	$Cell(a, v_1, Cw[id\|dir])$	$< DownReq_{wb}, a > \rightarrow dir$	$Cell(a, v_1, Tw[dir, (id, v)])$	M29
	$Cell(a, v_1, Tw[id\|dir, sm])$		$Cell(a, v_1, Tw[dir, sm\|(id, v)])$	M30
	$Cell(a, v_1, Cm[id])$		$Cell(a, v_1, Tw[\epsilon, (id, v)])$	M31
	$Cell(a, v_1, T'm[id])$		$Cell(a, v_1, Tw[\epsilon, (id, v)])$	M32
	$Cell(a, v_1, Tm[id, sm])$		$Cell(a, v_1, Tw[\epsilon, sm\|(id, v)])$	M33
$< Down_{wb}, a >$	$Cell(a, v, Cw[id\|dir])$		$Cell(a, v, Cw[dir])$	M34
	$Cell(a, v, Tw[id\|dir, sm])$		$Cell(a, v, Tw[dir, sm])$	M35
	$Cell(a, v, Cm[id])$		$Cell(a, v, Cw[\epsilon])$	M36
	$Cell(a, v, T'm[id])$		$Cell(a, v, Cw[\epsilon])$	M37
	$Cell(a, v, Tm[id, sm])$		$Cell(a, v, Tw[\epsilon, sm])$	M38
$< Down_{mw}, a >$	$Cell(a, v, Cm[id])$		$Cell(a, v, Cw[id])$	M39
	$Cell(a, v, T'm[id])$		$Cell(a, v, Cw[id])$	M40
	$Cell(a, v, Tm[id, sm])$		$Cell(a, v, Tw[id, sm])$	M41
$< DownV_{mw}, a, v >$	$Cell(a, v_1, Cm[id])$		$Cell(a, v, Cw[id])$	M42
	$Cell(a, v_1, T'm[id])$		$Cell(a, v, Cw[id])$	M43
	$Cell(a, v_1, Tm[id, sm])$		$Cell(a, v, Tw[id, sm])$	M44
$< Down_{mb}, a >$	$Cell(a, v, Cm[id])$		$Cell(a, v, Cw[\epsilon])$	M45
	$Cell(a, v, T'm[id])$		$Cell(a, v, Cw[\epsilon])$	M46
	$Cell(a, v, Tm[id, sm])$		$Cell(a, v, Tw[\epsilon, sm])$	M47
$< DownV_{mb}, a, v >$	$Cell(a, v_1, Cm[id])$		$Cell(a, v, Cw[\epsilon])$	M48
	$Cell(a, v_1, T'm[id])$		$Cell(a, v, Cw[\epsilon])$	M49
	$Cell(a, v_1, Tm[id, sm])$		$Cell(a, v, Tw[\epsilon, sm])$	M50
	$Cell(a, -, Tw[\epsilon, (id, v)\|sm])$	$< WbAck_b, a > \rightarrow id$	$Cell(a, v, Tw[\epsilon, sm])$	M51
	$Cell(a, -, Tw[\epsilon, (id, v)])$	$< WbAck_w, a > \rightarrow id$	$Cell(a, v, Tw[\epsilon, \epsilon])$	M52
	$Cell(a, -, Tw[\epsilon, (id, v)])$	$< WbAck_m, a > \rightarrow id$	$Cell(a, v, Tw[\epsilon, \epsilon])$	M53
	$Cell(a, v, Tw[\epsilon, \epsilon])$		$Cell(a, v, Cw[\epsilon])$	M54

Chapter 3

HWb: An Optimized Cachet protocol

In this chapter, we modify the Cachet protocol without changing the point-to-point message passing assumption. Most changes are motivated by a desire to make Cachet more suitable for buffer management, which we will discuss in the next Chapter. We call the modified Cachet - HWb. We offer some arguments about the correctness (Soundness) of the modified protocol but omit a full proof because it is long and tedious and not very insightful.

3.1 An overview of modifications to Cachet

We will make the following changes to Cachet:

1. We drop the distinction between the two types of writeback message: Wb_w and Wb_b, in Cachet because it is of no consequence. (See rules P20 and P22 in Table 2.3, C2 and C5 in Table 2.4, and M18-33 in Table 2.5).

2. We drop the rules for generating writeback acknowledgment messages, $WbAck_w$ and $WbAck_m$, because these are composite rules and can be derived from other rules. (See rules C30 and C31 in Table 2.4, M52 and M53 in Table 2.5).

3. Several rules in Cachet send multiple (invalidate-like) messages, one to each cache that has the location cached. (See rules M4, M18, M20, M28, M29 in Table 2.5). Blocking of one of the outgoing messages can lead to deadlock in a system with limited

communication buffer space. We replace such rules by finer-grained rules which inform one cache at a time. This creates a need for slightly finer-grain bookkeeping but does not necessarily increase the total storage requirements.

4. In Cachet, writeback messages that cannot be processed immediately, because the memory is busy invalidating the address in some caches, are stored in a suspension list. (See rules M18-33 in Table 2.5). Instead, we either leave the writeback messages in the incoming queue or negatively acknowledge it so that the affected cache can attempt the writeback later.

5. In Cachet, the value carried by a writeback message is temporarily stored in the suspended message buffer (See for example, rule M18-33 in Table 2.5) and it eventually replaces the value in the memory (See for example, rule M51-53 in Table 2.5). In case of several suspended writeback messages, the final writer is determined nondeterministically. Considerable space can be saved by immediately storing the writeback value in the memory and by just storing the id of the cache who has sent the Wb message (See for example, rules M18-27B in Table 3.3).

6. We assign higher priority to rules that make memory state transitions toward stable states than those that make transitions toward transient states. (For example, rules (M51, M54) over (M19) in Table 2.5).

3.2 The HWb Protocol

The specification of HWb, which incorporates all changes discussed above is shown in Table 3.1, Table 3.2, Table 3.3, and Table 3.4.

The following facts should be kept in mind while reading the new protocol:

- HWb has Wb instead of Wb_b and Wb_w message types.

- The suspended message buffer in HWb (GM) contains the identifier of the cache that sent the writeback message. This is different from the suspended message buffer in Cachet (SM), which contains both the identifiers and the values of writeback messages.

- In HWb, the transient state Tw has two directories while in Cachet, Tw state has one directory. The first directory is the same as the one in Cachet and contains the ids

of caches that have the address in the Writer-Push state and to whom $DownReq_{wb}$ message has been sent but no reply has been received. The second directory contains ids of caches that have the address in Writer-Push states and to whom $DownReq_{wb}$ message has not been sent.

As a consequence of voluntary downgrading rules (rules C3 and C4 in Table 2.4), the memory may receive an (implicit) response to a $DownReq_{wb}$ from a cache even before it sends the $DownReq_{wb}$ message to that cache. In this situation, we simply drop the appropriate id from the second directory (See rule M2B in Table 3.4).

Table 3.1: HWb: The Processor Rules

Mandatory Processor Rules				
Instruction	**Cstate**	**Action**	**Next Cstate**	
$Loadl(a)$	$Cell(a, v, Clean_b)$	*retire*	$Cell(a, v, Clean_b)$	P1
	$Cell(a, v, Dirty_b)$	*retire*	$Cell(a, v, Dirty_b)$	P2
	$Cell(a, v, Clean_w)$	*retire*	$Cell(a, v, Clean_w)$	P3
	$Cell(a, v, Dirty_w)$	*retire*	$Cell(a, v, Dirty_w)$	P4
	$Cell(a, v, Clean_m)$	*retire*	$Cell(a, v, Clean_m)$	P5
	$Cell(a, v, Dirty_m)$	*retire*	$Cell(a, v, Dirty_m)$	P6
	$Cell(a, v, WbPending)$	*stall*	$Cell(a, v, WbPending)$	P7
	$Cell(a, -, CachePending)$	*stall*	$Cell(a, -, CachePending)$	P8
	$a \notin cache$	$stall, < CacheReq, a > \rightarrow H$	$Cell(a, -, CachePending)$	P9
$Storel(a, v)$	$Cell(a, v, Clean_b)$	*retire*	$Cell(a, v, Dirty_b)$	P10
	$Cell(a, v, Dirty_b)$	*retire*	$Cell(a, v, Dirty_b)$	P11
	$Cell(a, v, Clean_w)$	*retire*	$Cell(a, v, Dirty_w)$	P12
	$Cell(a, v, Dirty_w)$	*retire*	$Cell(a, v, Dirty_w)$	P13
	$Cell(a, v, Clean_m)$	*retire*	$Cell(a, v, Dirty_m)$	P14
	$Cell(a, v, Dirty_m)$	*retire*	$Cell(a, v, Dirty_m)$	P15
	$Cell(a, v_1, WbPending)$	*stall*	$Cell(a, v_1, WbPending)$	P16
	$Cell(a, -, CachePending)$	*stall*	$Cell(a, -, CachePending)$	P17
	$a \notin cache$	$stall, < CacheReq, a > \rightarrow H$	$Cell(a, -, CachePending)$	P18
$Commit(a)$	$Cell(a, v, Clean_b)$	*retire*	$Cell(a, v, Clean_b)$	P19
	$Cell(a, v, Dirty_b)$	$stall, < Wb, a, v > \rightarrow H$	$Cell(a, v, WbPending)$	P20
	$Cell(a, v, Clean_w)$	*retire*	$Cell(a, v, Clean_w)$	P21
	$Cell(a, v, Dirty_w)$	$stall, < Wb, a, v > \rightarrow H$	$Cell(a, v, WbPending)$	P22
	$Cell(a, v, Clean_m)$	*retire*	$Cell(a, v, Clean_m)$	P23
	$Cell(a, v, Dirty_m)$	*retire*	$Cell(a, v, Dirty_m)$	P24
	$Cell(a, v, WbPending)$	*stall*	$Cell(a, v, WbPending)$	P25
	$Cell(a, -, CachePending)$	*stall*	$Cell(a, -, CachePending)$	P26
	$a \notin cache$	*retire*	$a \notin cache$	P27
$Reconcile(a)$	$Cell(a, -, Clean_b)$	*retire*	$a \notin cache$	P28
	$Cell(a, v, Dirty_b)$	*retire*	$Cell(a, v, Dirty_b)$	P29

Continued on next page

Table 3.1 – continued from previous page

Instruction	Cstate	Action	Next Cstate	
	$Cell(a, v, Clean_w)$	*retire*	$Cell(a, v, Clean_w)$	P30
	$Cell(a, v, Dirty_w)$	*retire*	$Cell(a, v, Dirty_w)$	P31
	$Cell(a, v, Clean_m)$	*retire*	$Cell(a, v, Clean_m)$	P32
	$Cell(a, v, Dirty_m)$	*retire*	$Cell(a, v, Dirty_m)$	P33
	$Cell(a, v, WbPending)$	*stall*	$Cell(a, v, WbPending)$	P34
	$Cell(a, -, CachePending)$	*stall*	$Cell(a, -, CachePending)$	P35
	$a \notin cache$	*retire*	$a \notin cache$	P36

Table 3.1: HWb: The Cache Engine Rules

Voluntary C-engine Rules				
	Cstate	**Action**	**Next Cstate**	
	$Cell(a, -, Clean_b)$		$a \notin cache$	C1
	$Cell(a, v, Dirty_b)$	$< Wb, a, v > \rightarrow H$	$Cell(a, v, WbPending)$	C2
	$Cell(a, v, Clean_w)$	$< Down_{wb}, a > \rightarrow H$	$Cell(a, v, Clean_b)$	C3
	$Cell(a, v, Dirty_w)$	$< Down_{wb}, a > \rightarrow H$	$Cell(a, v, Dirty_b)$	C4
		$< Wb, a, v > \rightarrow H$	$Cell(a, v, WbPending)$	C5
	$Cell(a, v, Clean_m)$	$< Down_{mw}, a > \rightarrow H$	$Cell(a, v, Clean_w)$	C6
		$< Down_{mb}, a > \rightarrow H$	$Cell(a, v, Clean_b)$	C7
	$Cell(a, v, Dirty_m)$	$< DownV_{mw}, a, v > \rightarrow H$	$Cell(a, v, Clean_w)$	C8
		$< DownV_{mb}, a, v > \rightarrow H$	$Cell(a, v, Clean_b)$	C9
	$a \notin cache$	$< CacheReq, a > \rightarrow H$	$Cell(a, -, CachePending)$	C10

Mandatory C-engine Rules				
Msg from H	**Cstate**	**Action**	**Next Cstate**	
$< Cache_b, a, v >$	$Cell(a, -, CachePending)$		$Cell(a, v, Clean_b)$	C11
$< Cache_w, a, v >$	$Cell(a, -, Clean_b)$		$Cell(a, v, Clean_w)$	C12
	$Cell(a, v_1, Dirty_b)$		$Cell(a, v_1, Dirty_w)$	C13
	$Cell(a, v_1, WbPending)$		$Cell(a, v_1, WbPending)$	C14
	$Cell(a, -, CachePending)$		$Cell(a, v, Clean_w)$	C15
	$a \notin cache$		$Cell(a, v, Clean_w)$	C16
$< Cache_m, a, v >$	$Cell(a, -, Clean_b)$		$Cell(a, v, Clean_m)$	C17
	$Cell(a, v_1, Dirty_b)$		$Cell(a, v_1, Dirty_m)$	C18
	$Cell(a, v_1, WbPending)$		$Cell(a, v_1, WbPending)$	C19
	$Cell(a, -, CachePending)$		$Cell(a, v, Clean_m)$	C20
	$a \notin cache$		$Cell(a, v, Clean_m)$	C21
$< Up_{wm}, a >$	$Cell(a, v, Clean_b)$		$Cell(a, v, Clean_b)$	C22
	$Cell(a, v, Dirty_b)$		$Cell(a, v, Dirty_b)$	C23
	$Cell(a, v, Clean_w)$		$Cell(a, v, Clean_m)$	C24
	$Cell(a, v, Dirty_w)$		$Cell(a, v, Dirty_m)$	C25
	$Cell(a, v, WbPending)$		$Cell(a, v, WbPending)$	C26

Continued on next page

Table 3.2 – continued from previous page

Msg from H	Cstate	Action	Next Cstate	
	$Cell(a, -, CachePending)$		$Cell(a, -, CachePending)$	C27
	$a \notin cache$		$a \notin cache$	C28
$< WbAck_b, a >$	$Cell(a, v, WbPending)$		$Cell(a, v, Clean_b)$	C29
$< DownReq_{wb}, a >$	$Cell(a, v, Clean_b)$		$Cell(a, v, Clean_b)$	C30
	$Cell(a, v, Dirty_b)$		$Cell(a, v, Dirty_b)$	C31
	$Cell(a, v, Clean_w)$	$< Down_{wb}, a > \rightarrow H$	$Cell(a, v, Clean_b)$	C32
	$Cell(a, v, Dirty_w)$	$< Down_{wb}, a > \rightarrow H$	$Cell(a, v, Dirty_b)$	C33
	$Cell(a, v, WbPending)$		$Cell(a, v, WbPending)$	C34
	$Cell(a, -, CachePending)$		$Cell(a, -, CachePending)$	C35
	$a \notin cache$		$a \notin cache$	C36
$< DownReq_{mw}, a >$	$Cell(a, v, Clean_b)$		$Cell(a, v, Clean_b)$	C37
	$Cell(a, v, Dirty_b)$		$Cell(a, v, Dirty_b)$	C38
	$Cell(a, v, Clean_w)$		$Cell(a, v, Clean_w)$	C39
	$Cell(a, v, Dirty_w)$		$Cell(a, v, Dirty_w)$	C40
	$Cell(a, v, Clean_m)$	$< Down_{mw}, a > \rightarrow H$	$Cell(a, v, Clean_w)$	C41
	$Cell(a, v, Dirty_m)$	$< DownV_{mw}, a, v > \rightarrow H$	$Cell(a, v, Clean_w)$	C42
	$Cell(a, v, WbPending)$		$Cell(a, v, WbPending)$	C43
	$Cell(a, -, CachePending)$		$Cell(a, -, CachePending)$	C44
	$a \notin cache$		$a \notin cache$	C45
$< DownReq_{mb}, a >$	$Cell(a, v, Clean_b)$		$Cell(a, v, Clean_b)$	C46
	$Cell(a, v, Dirty_b)$		$Cell(a, v, Dirty_b)$	C47
	$Cell(a, v, Clean_w)$	$< Down_{wb}, a > \rightarrow H$	$Cell(a, v, Clean_b)$	C48
	$Cell(a, v, Dirty_w)$	$< Down_{wb}, a > \rightarrow H$	$Cell(a, v, Dirty_b)$	C49
	$Cell(a, v, Clean_m)$	$< Down_{mb}, a > \rightarrow H$	$Cell(a, v, Clean_b)$	C50
	$Cell(a, v, Dirty_m)$	$< DownV_{mb}, a, v > \rightarrow H$	$Cell(a, v, Clean_b)$	C51
	$Cell(a, v, WbPending)$		$Cell(a, v, WbPending)$	C52
	$Cell(a, -, CachePending)$		$Cell(a, -, CachePending)$	C53
	$a \notin cache$		$a \notin cache$	C54

Table 3.3: HWb: The Memory Engine Rules-A

Voluntary M-engine Rules				
	Mstate	Action	Next Mstate	
	$Cell(a, v, Cw[dir])(id \notin dir)$	$< Cache_w, a, v > \rightarrow id$	$Cell(a, v, Cw[id \mid dir])$	M1
	$Cell(a, v, Cw[id])$	$< Up_{wm}, a > \rightarrow id$	$Cell(a, v, Cm[id])$	M2
	$Cell(a, v, Cw[\epsilon])$	$< Cache_m, a, v > \rightarrow id$	$Cell(a, v, Cm[id])$	M3
	$Cell(a, v, Cw[id \mid dir])$	$< DownReq_{wb}, a > \rightarrow id$	$Cell(a, v, Tw[id, dir, \epsilon])$	M4
	$Cell(a, v, Cm[id])$	$< DownReq_{mw}, a > \rightarrow id$	$Cell(a, v, T'm[id])$	M5
		$< DownReq_{mb}, a > \rightarrow id$	$Cell(a, v, Tm[id, \epsilon])$	M6
	$Cell(a, v, T'm[id])$	$< DownReq_{wb}, a > \rightarrow id$	$Cell(a, v, Tm[id, \epsilon])$	M7

Continued on next page

Table 3.3 – continued from previous page

Mandatory M-engine Rules				
Msg from id	**Mstate**	**Action**	**Next Mstate**	
$< CacheReq, a >$	$Cell(a, v, Cw[dir])(id \notin dir)$	$< Cache_b, a, v > \rightarrow id$	$Cell(a, v, Cw[dir])$	M8
	$Cell(a, v, Tw[dir_1, dir_2, gm])(id \notin dir_1 \mid dir_2)$	*stall message*	$Cell(a, v, Tw[dir_1, dir_2, , gm])$	M9
	$Cell(a, v, Cw[dir])(id \in dir)$		$Cell(a, v, Cw[dir])$	M10
	$Cell(a, v, Tw[dir_1, dir_2, gm])(id \in dir_1 \mid dir_2)$		$Cell(a, v, Tw[dir_1, dir_2, gm])$	M11
	$Cell(a, v, Cm[id_1])(id \neq id_1)$	*stall message* $< DownReq_{mw}, a > \rightarrow id_1$	$Cell(a, v, T'm[id_1])$	M12
	$Cell(a, v, T'm[id_1])(id \neq id_1)$	*stall message*	$Cell(a, v, T'm[id_1])$	M13
	$Cell(a, v, Tm[id_1, gm])(id \neq id_1)$	*stall message*	$Cell(a, v, Tm[id_1, gm])$	M14
	$Cell(a, v, Cm[id])$		$Cell(a, v, Cm[id])$	M15
	$Cell(a, v, T'm[id])$		$Cell(a, v, T'm[id])$	M16
	$Cell(a, v, Tm[id, gm])$		$Cell(a, v, Tm[id, gm])$	M17
$< Wb, a, v >$	$Cell(a, v_1, Cw[dir])(id \notin dir)$		$Cell(a, v, Tw[\epsilon, dir, id])$	M18
	$Cell(a, v_1, Tw[dir_1, dir_2, gm])(id \notin dir_1 \mid dir_2, dir_1 \mid dir_2 \neq \epsilon)$		$Cell(a, v, Tw[dir_1, dir_2 gm \mid id])$	M19A
	$Cell(a, v_1, Tw[\epsilon, \epsilon, gm])$	*stall message*	$Cell(a, v_1, Tw[\epsilon, \epsilon, gm])$	M19B
	$Cell(a, v_1, Cw[id \mid dir])$		$Cell(a, v, Tw[\epsilon, dir, id])$	M20A
	$Cell(a, v_1, Cw[id \mid dir])$	*stall message*	$Cell(a, v_1, Tw[\epsilon, dir, \epsilon])$	M20B
	$Cell(a, v_1, Tw[id \mid dir_1, dir_2, gm])$		$Cell(a, v, Tw[dir_1, dir_2, gm \mid id])$	M21A1
	$Cell(a, v_1, Tw[dir_1, id \mid dir_2, gm])$		$Cell(a, v, Tw[dir_1, dir_2, gm \mid id])$	M21A2
	$Cell(a, v_1, Tw[id \mid dir_1, dir_2, gm])$	*stall message*	$Cell(a, v_1, Tw[dir_1, dir_2, gm])$	M21B1
	$Cell(a, v_1, Tw[dir_1, id \mid dir_2, gm])$	*stall message*	$Cell(a, v_1, Tw[dir_1, dir_2, gm])$	M21B2
	$Cell(a, v_1, Cm[id_1])(id \neq id_1)$	$< DownReq_{mb}, a > \rightarrow id_1$	$Cell(a, v, Tm[id_1, id])$	M22
	$Cell(a, v_1, T'm[id_1])(id \neq id_1)$	$< DownReq_{wb}, a > \rightarrow id_1$	$Cell(a, v, Tm[id_1, id])$	M23
	$Cell(a, v_1, Tm[id_1, gm])(id \neq id_1)$		$Cell(a, v, Tm[id_1, gm \mid id])$	M24
	$Cell(a, v_1, Cm[id])$		$Cell(a, v, Tw[\epsilon, \epsilon, id])$	M25A
	$Cell(a, v_1, Cm[id])$	*stall message*	$Cell(a, v_1, Tw[\epsilon, \epsilon, \epsilon])$	M25B
	$Cell(a, v_1, T'm[id])$		$Cell(a, v, Tw[\epsilon, \epsilon, id])$	M26A
	$Cell(a, v_1, T'm[id])$	*stall message*	$Cell(a, v_1, Tw[\epsilon, \epsilon, \epsilon])$	M26B
	$Cell(a, v_1, Tm[id, gm])$		$Cell(a, v, Tw[\epsilon, \epsilon, gm \mid id])$	M27A
	$Cell(a, v_1, Tm[id, gm])$	*stall message*	$Cell(a, v_1, Tw[\epsilon, \epsilon, gm])$	M27B

Table 3.4: HWb: The Memory Engine Rules-B

Mandatory M-engine Rules				
Msg from id	**Mstate**	**Action**	**Next Mstate**	
$< Down_{wb}, a >$	$Cell(a, v, Cw[id\|dir])$		$Cell(a, v, Cw[dir])$	M1
	$Cell(a, v, Tw[id\|dir_1, dir_2, gm])$		$Cell(a, v, Tw[dir_1, dir_2, gm])$	M2A
	$Cell(a, v, Tw[dir_1, id\|dir_2, gm])$		$Cell(a, v, Tw[dir_1, dir_2, gm])$	M2B
	$Cell(a, v, Cm[id])$		$Cell(a, v, Cw[\epsilon])$	M3
	$Cell(a, v, T'm[id])$		$Cell(a, v, Cw[\epsilon])$	M4
	$Cell(a, v, Tm[id, gm])$		$Cell(a, v, Tw[\epsilon, \epsilon, gm])$	M5
$< Down_{mw}, a >$	$Cell(a, v, Cm[id])$		$Cell(a, v, Cw[id])$	M6
	$Cell(a, v, T'm[id])$		$Cell(a, v, Cw[id])$	M7
	$Cell(a, v, Tm[id, gm])$		$Cell(a, v, Tw[id, \epsilon, gm])$	M8
$< DownV_{mw}, a, v >$	$Cell(a, v_1, Cm[id])$		$Cell(a, v, Cw[id])$	M9
	$Cell(a, v_1, T'm[id])$		$Cell(a, v, Cw[id])$	M10
	$Cell(a, v_1, Tm[id, \epsilon])$		$Cell(a, v, Tw[id, \epsilon, \epsilon])$	M11A
	$Cell(a, v_1, Tm[id, gm])(gm \neq \epsilon)$		$Cell(a, v_1, Tw[id, \epsilon, gm])$	M11B
$< Down_{mb}, a >$	$Cell(a, v, Cm[id])$		$Cell(a, v, Cw[\epsilon])$	M12
	$Cell(a, v, T'm[id])$		$Cell(a, v, Cw[\epsilon])$	M13
	$Cell(a, v, Tm[id, gm])$		$Cell(a, v, Tw[\epsilon, \epsilon, gm])$	M14
$< DownV_{mb}, a, v >$	$Cell(a, v_1, Cm[id])$		$Cell(a, v, Cw[\epsilon])$	M15
	$Cell(a, v_1, T'm[id])$		$Cell(a, v, Cw[\epsilon])$	M16
	$Cell(a, v_1, Tm[id, \epsilon])$		$Cell(a, v, Tw[\epsilon, \epsilon, \epsilon])$	M17A
	$Cell(a, v_1, Tm[id, gm])(gm \neq \epsilon)$		$Cell(a, v_1, Tw[\epsilon, \epsilon, gm])$	M17B
	$Cell(a, v, Tw[\epsilon, \epsilon, id\|gm])$	$< WbAck_b, a > \rightarrow id$	$Cell(a, v, Tw[\epsilon, \epsilon, gm])$	M18
	$Cell(a, v, Tw[\epsilon, \epsilon, \epsilon])$		$Cell(a, v, Cw[\epsilon])$	M19
	$Cell(a, v, Tw[dir_1, id\|dir_2, gm])$	$< DownReq_{wb}, a > \rightarrow id$	$Cell(a, v, Tw[id\|dir_1, dir_2, gm])$	MDir1

3.3 Soundness of HWb

We argue the correctness of each modification discussed in Section 3.1:

1. The rules on the memory side corresponding to message types Wb_w and Wb_b (i.e., rules M18-33 in Table 2.5) produce outputs which are identical. Hence the difference between Wb_w and Wb_b is inconsequential.

2. The effect of the rule M52 in Table 2.5 can be achieved by firing M51 and M1. The effect of M53 in Table 2.5 can be achieved by firing M51 and M3. Similarly, the effect of C30 in Table 2.4 can be achieved by C29 and C12, and the effect of C31 by C29

and C17. Consequently, all these are derived rules.

3. Sending out downgrading messages one by one instead of in a bulk does not affect the soundness of the protocol because the combination of the first and the second directories in Tw state of HWb is equivalent to the corresponding directory in the Tw state of Cachet. The main difference between the two protocols is that in case a Down message or an implicit acknowledgment in the form of a Wb message is received from a cache even before the downgrading request is sent to the cache, HWb does not even sent downgrading request. In case the downgrading request has been sent the cache simply ignores it.

4. Once a writeback message is suspended in sm in Cachet, it has no effect on the processing of incoming messages until the memory state becomes non-transient. Hence, it should not matter whether the suspended message buffer is tied to the address line or simply waiting in the input queue. The only subtlety is that because of the fine-grain id-deletion in Wb processing, the id of the cache sending the writeback may be the one who either has not been sent an downgrade request or has not processed such a request. In either case such a request will be ignored by the cache if and when the message reaches the memory. Our protocol is sound because we erase id of the stalled writeback message from the (first and second) directory.

5. Immediately storing the writeback value in the memory is sound because memory value in a transient memory state of Cachet cannot be read by caches until the state of the address becomes stable again. (See for example, rules M51-53 in Table 2.5). In HWb, we can save the writeback value in the memory at the moment of receiving the writeback message because writing the value will eventually be done and the previous value will never be used again. (See M18-27B in Table 3.3).

6. Assigning higher priority to a rule (that make memory state transitions toward stable states than those that make transitions toward transient states) cannot affect the soundness of the protocol because we make the system more deterministic by adding more conditions under which which actions will be taken. This may affect the liveness, however. We will show that the liveness of the final protocol (BCachet), which is based on HWb, in Chapter 5.

Chapter 4

BCachet: Cachet with Buffer Management

The optimized version of Cachet, HWb, that we presented in Chapter 3 uses the same buffer management scheme as Cachet. The buffer scheme that is implied by Cachet communication rules (see Figure 2-4) leads to $N \times P$ FIFOs for a system with N address lines and P processors. Both Cachet and HWb require space for suspended messages whose processing has to be deferred until later. Cachet stores such messages in a suspended message list (sm) in every location, while HWb can leave them in the external input buffer until the suspended message list has a space for messages. Suspended messages have to be stored in a manner that does not block the incoming traffic in a way that creates deadlocks.

In this chapter, we will modify HWb to produce BCachet such that BCachet can be implemented using only two FIFOs per memory unit and which has a fixed and small buffer requirement for suspended messages.

4.1 The BCachet protocol

BCachet has high priority and low priority FIFOs from processors to memory. The type of a message determines if it should be enqueued in a high priority(H) or a low priority(L) FIFO. The messages from the memory to a processor share a single queue. A special queue (STQ) is provided at the memory end to hold stalled messages. Please see Figure 4-1 and 4-2 for the BCachet system configuration. The salient features of BCachet are described next:

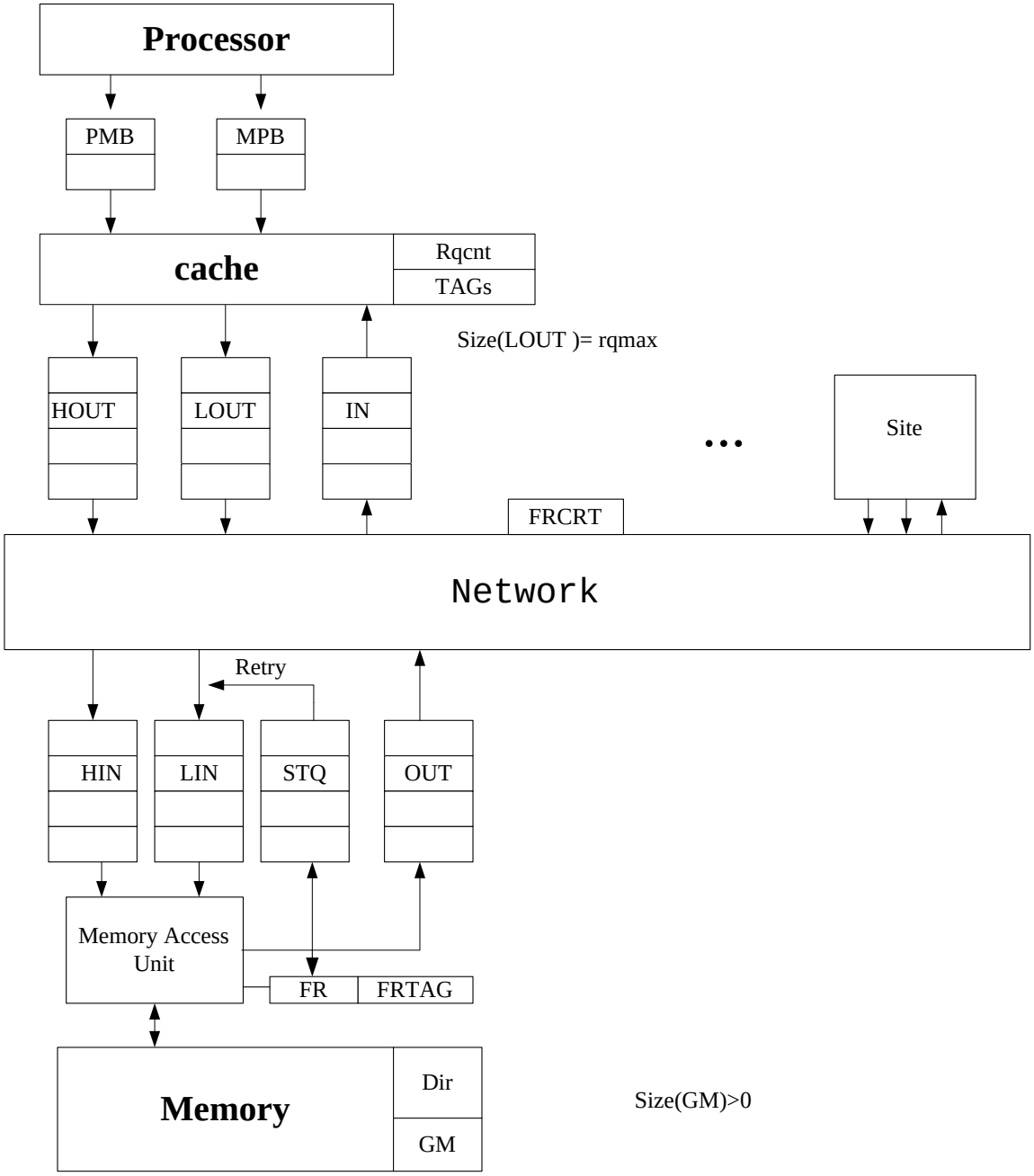

Figure 4-1: Buffer Management System (BCachet) Overview

SYS	$\equiv$	Sys(MSITE,SITEs)	*System*
MSITE	$\equiv$	Msite(MEM,HIN,LIN,STQ,FR,OUT, FRTAG,FRCRT)	*Memory Site*
FR	$\equiv$	ϵ [] MSG	*First Priority Request Register*
FRTAG	$\equiv$	ϵ [] (id,a)	*FR Tag*
FRCRT	$\equiv$	ϵ [] id	*Fairness Controller*
STQ	$\equiv$	ϵ [] MSG;STQ	*Stalled Message Queue*
MEM	$\equiv$	ϵ [] Cell(a,v,MSTATE)|MEM	*Memory*
SITEs	$\equiv$	SITE [] SITE|SITEs	*Set of Cache Sites*
SITE	$\equiv$	Site(id,CACHE,rqcnt,TAGs,IN,HOUT,LOUT, PMB,MPB,PROC)	*Cache Site*
CACHE	$\equiv$	ϵ [] Cell(a,v,CSTATE)|CACHE	*Cache*
TAGs	$\equiv$	ϵ [] t|TAGs	*Request Tag*
IN	$\equiv$	ϵ [] MSG;IN	*Incoming Queue*
HIN	$\equiv$	ϵ [] MSG	*High Priority Incoming Queue*
LIN	$\equiv$	ϵ [] MSG;LIN	*Low Priority Incoming Queue*
OUT	$\equiv$	ϵ [] MSG;OUT	*Outgoing Queue*
HOUT	$\equiv$	ϵ [] MSG;HOUT	*High Priority Outgoing Queue*
LOUT	$\equiv$	ϵ [] MSG;LOUT	*Low Priority Outgoing Queue*
MSG	$\equiv$	Msg(src,$dest$,CMD,a,Value)	*Message*
Value	$\equiv$	ϵ [] v	*Value*
MSTATE	$\equiv$	Cw[DIR] [] Tw[DIR,DIR,GM] [] Cm[id] [] Tm[id,GM] [] T'm[id]	*Memory State*
DIR	$\equiv$	ϵ [] id|DIR	*Directory*
GM	$\equiv$	ϵ [] id|GM	*Suspended Message*
CSTATE	$\equiv$	Clean_b [] Dirty_b [] Clean_w [] Dirty_w [] Clean_m [] Dirty_m [] WbPending [] CachePending [] (Clean_b,Down_{wb}) [] (Clean_b,Down_{mb}) [] (Clean_w,ϵ) [] (Clean_w,Down_{mw}) [] (Clean_m,ϵ) []	*Cache State*
CMD	$\equiv$	CacheReq [] Wb [] Down_{wb} [] Down_{mw} [] DownV_{mw} [] Down_{mb} [] DownV_{mb} [] Cache_w [] Cache_m [] Up_{wm} [] WbAck_b [] DownReq_{wb} [] DownReq_{mw} [] DownReq_{mb} [] CacheAck [] CacheNack [] WbNack [] ErFRTag	*Command*
One Path	:	Cache_w, Cache_m, Up_{wm}, WbAck_b, DownReq_{mb}, DownReq_{mw}, DownReq_{wb}, CacheAck, CacheNack, WbNack	
High Priority	:	Down_{mw}, Down_{wb}, Down_{mb}, DownV_{mw},	
	:	DownV_{mb}, ErFRTag	
Low Priority	:	CacheReq, Wb	
rq_{max}	:		*Maximum Number of Request per Site*
P	:		*Number of Cache Sites*
$Size(HIN)$	$\geq$	1	*Memory's High Incoming Buffer Size*
$Size(LIN)$	$\geq$	1	*Memory's Low Incoming Buffer Size*
$Size(OUT)$	$\geq$	2	*Memory's Outgoing Buffer Size*
$Size(HOUT)$	$\geq$	2	*Cache's High Outgoing Buffer Size*
$Size(LOUT)$	$\geq$	Rq_{max}	*Cache's Low Outgoing Buffer Size*
$Size(IN)$	$\geq$	1	*Cache's Incoming Buffer Size*
$Size(STQ)$	$\geq$	0	*Stalled Message Queue Size*
$Size(Dir)$	$\geq$	0	*Directory Size*
$Size(GM)$	$\geq$	1	*Suspended Message Buffer Size*

Figure 4-2: TRS expression of BCachet

1. **Request Counter**: In BCachet, we introduce a request counter (Rqcnt) per cache to control the number of outstanding request messages (i.e., Wb, CacheReq) from the cache. A cache increases its Rqcnt by one when it sends a request type message and decreases its Rqcnt by one when it receives a response. When a Rqcnt reaches the maximum value, rq_{max}, it postpones sending a request message until Rqcnt becomes less than rq_{max}.

2. **High and Low Message Paths**: A BCachet system uses high priority and low priority FIFOs for cache-to-memory messages instead of a point-to-point buffers. When a cache sends a message, it is determined by the type of the message whether it should be enqueued in a high priority(H) or a low priority(L) FIFO. Every response type message uses H-FIFO (i.e., Down, DownV, ErFrTag) while every request type message uses L-FIFO (i.e.,CaheReq, Wb). (See Figure 4-2). For memory-to-cache messages, a BCachet system has only one FIFO path. (See Figure 4-2).

3. **Message Holding (Locked Cache States & CacheAck Message)**: Separation of message path based on the message type is unsound with respect to point-to-point message passing. The point-to-point message passing does not allow two messages to be reordered if they have the same address, source, and destination. However, two separate paths may reorder two messages although they have the same address, source and destination in case the front message is CacheReq and the following message is a high priority message. The scenario is shown in Figure 4-3. Let say msg_1 is a CacheReq message and msg_2 is a high priority message. First, msg_1 goes into LFIFO and it waits to proceed. Before msg_1 proceeds, msg_2 enters HFIFO. Since msg_2 is in HFIFO, it proceeds first. Finally, msg_1 proceeds. The order of these two messages is switched, even though msg_1 and msg_2 have the same address, source, and destination. We call this reordering of messages - incorrect reordering.

   We introduce message holding scheme to solve the incorrect reordering problem. In BCachet, a cache dose not send a message if the message may cause incorrect reordering. Instead of sending a dangerous message, a cache block stores the information of the message in a cache state until sending it becomes safe. We call this scheme

message holding.

For message holding scheme, we introduce five locked cache states, (Clean_b,Down_{wb}), (Clean_b,Down_{mb}), (Clean_w, ϵ), (Clean_w,Down_{mw}), and (Clean_m,ϵ) as well as CacheAck message. A locked cache state is a tuple of HWb's cache state and message, and thus, indicates that a cache block is in the cache state and the message is held by the cache block. For example, a cache block in (Clean_b,Down_{wb}) indicates that the cache block is Clean_b state a $Down_{wb}$ message is held in the cache.

A cache block enters locked cache states from CachePending when the cache block receives $Cache_w$ or $Cache_m$ message. In HWb, when a cache sends a CacheReq message, the cache sets the block of the address as CachePending. It is possible that the cache receives $Cache_w$ or $Cache_m$ sent by voluntary cache rules (See M1 and M3 in Table 3.3) and sets the block as $Clean_w$ or $Clean_m$ (See C15 and C20 in Table 3.2) before the CacheReq message disappears in the buffer. In BCachet, instead of setting the cache state as $Cache_w$ or $Cache_m$, the cache sets the cache state as ($Cache_w$,ϵ) or ($Cache_m$,ϵ), respectively (See MC4 and MC9 in Table 4.3).

Once a cache block enters locked cache states, sending a message is not allowed until the CacheReq sent by the block disappears in the system. It is possible that the cache receives request for downgrading, before the CacheReq disappears in the system. In that case, the cache can response to the downgrading request by sets the cache state as an appropriate locked cache state. (See MC28-MC31, MC41-MC45, and MC55-MC59 in Table 4.3). Execution of some instructions and protocol changes in some situations are allowed by the same way. (See P10-P14, P24-P28, P38-P42, P52-P56 in Table 4.1,

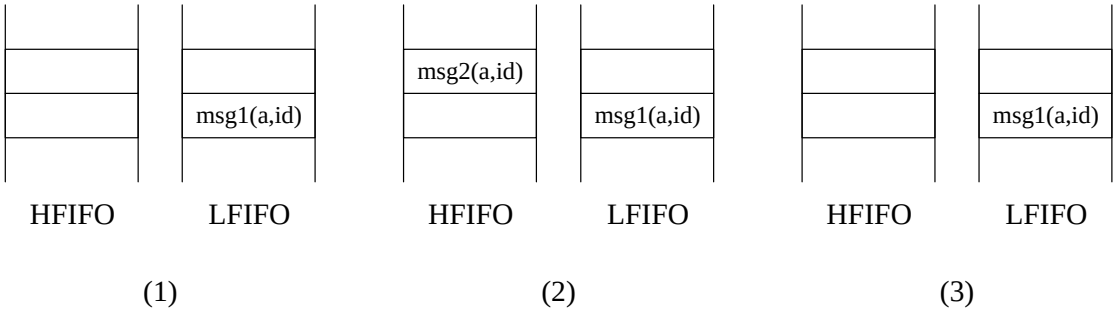

Figure 4-3: Incorrect Reordering

VC11-VC14 in Table 4.2, MC18-MC19, MC28-MC31, MC41-MC45, MC55-MC59 in Table 4.3).

The locked cache states become normal cache states when the cache block is informed that the CacheReq sent by the block disappeared in the system. CacheAck message is used to inform the cache of disappearance of the CacheReq message. When a CacheReq is served, the memory sends CacheAck to the sender cache so that the sender cache of the CacheReq knows that sending the message for the address becomes safe. (See FRC1-FRC6 in Table 4.6, CHA1, CHA3, CHA4, CHA8-CHA10 in Table 4.7). On receiving a CacheAck message, the cache sets the cache state as the corresponding normal state and releases a held message if the initial cache state contains a held message. (See MC60-MC65 in Table 4.3).

Figure 4-4 shows an cache state transition related to locked cache states in BCachet and Figure 4-5 shows the complete cache state transition diagram in HWb. We can easily check that the locked cache state transitions can be simulated by cache state transitions in HWb. For example, a cache state transition from $(Clean_w, \epsilon)$ to $(Clean_b, Down_{wb})$ in BCachet can be thought as a cache state transition $Clean_w$ to $Clean_b$ and the action, sending a $Down_{wb}$ message.

4. **Negative Acknowledgment**: We use negative acknowledgment method to distribute the message congestion over sites in case too many request messages are sent to the memory. A BCachet's memory site temporarily stores a request message in stalled message queue (STQ) if the request cannot be served. If the stalled message queue is full, then the memory sends the corresponding Nack signal back to the sender of the request message so that the sender resend the request again. (See for example, SNM2, SNM4 in Table 4.7, MC66, MC67 in Table 4.3). It is guaranteed that each cache site can handle all request messages generated by it and the Nack signals sent to it because the size of cache's outgoing low priority message queue (LOUT) must be equal to or larger than the maximum number of request message generated by the cache (See Figure 4-2).

Although the size of STQ does not affect the liveness of the system, it affects system's

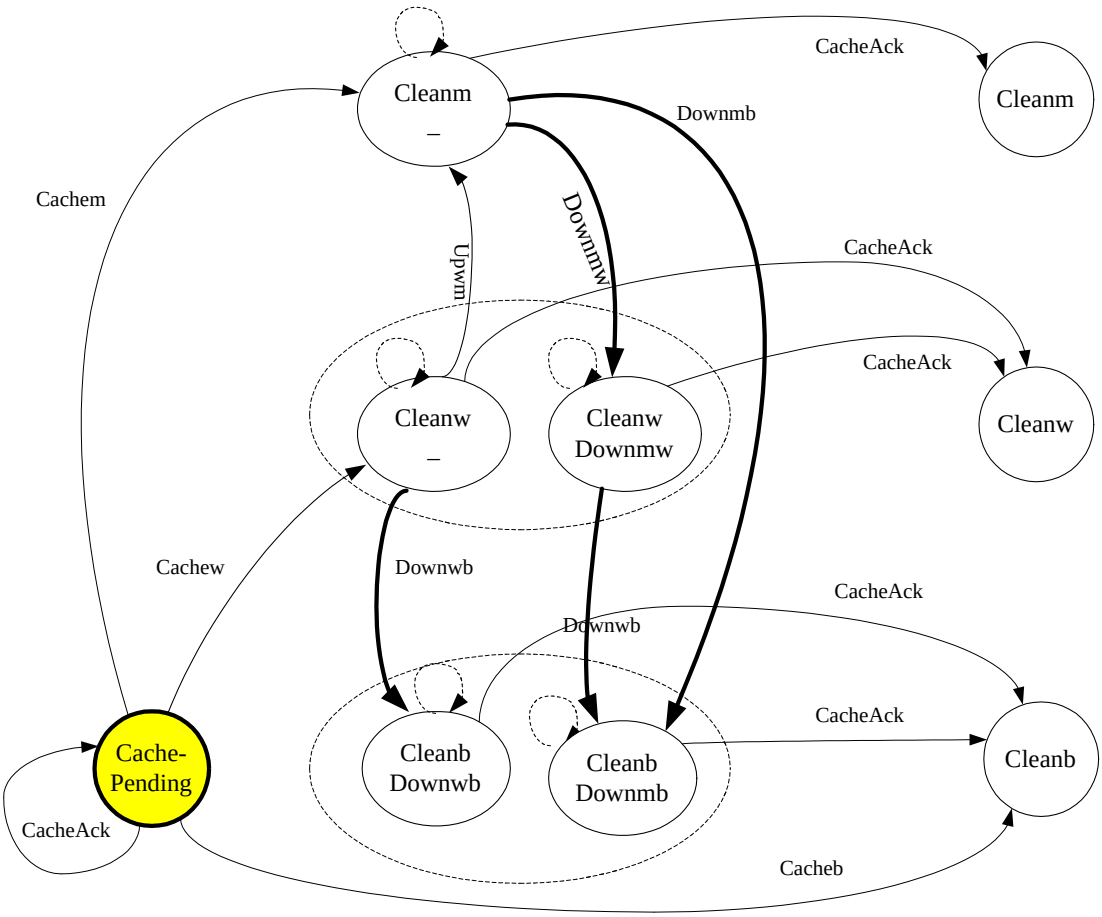

Figure 4-4: Locked Cache State Transition Diagram

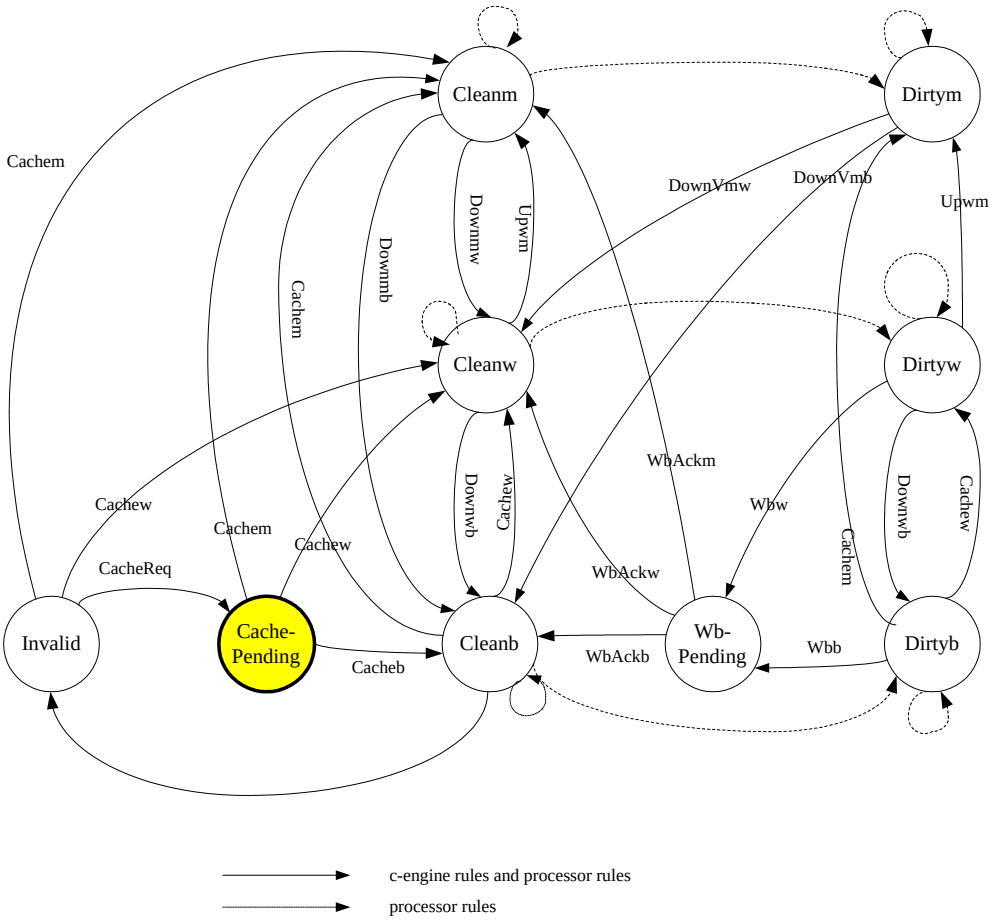

Figure 4-5: Complete Cache State Transition Diagram of the Previous Protocol

performance. If STQ is too small, then the memory engine may send many Nack signals. Therefore, the systems is likely to consume consumes much bandwidth for Nack signals.

5. **Directory and Suspended Message Buffer**:

   A directory (Dir) and a suspended buffer (GM) of a memory are reserved memory space shared by all address lines. Note that we are not assuming each memory block has its own Dir and GM. (See Figure 4-6). Directory size does not affect the liveness of Cachet, because of adaptive mechanism of protocol change in Cachet and the nature of the base protocol that the base protocol does not require directory. The liveness of the system is guaranteed as long as the size of GM is greater than or equal to one name tag of a processor (See Figure 4-2).

   The size of Dir and GM may affect a lot system's performance. If Dir is too small, then lots of cache blocks cannot upgrade to Writer-Push or Migratory protocol. If GM is too small, then the memory only can serve one Wb message at a time, and thus, other Wb messages must wait in STQ or be negatively acknowledged.

6. **Fairness Control**: In BCachet, fairness units are used to to avoid livelock. BCachet uses a first priority request register (FR), a first priority request register tag (FRTag), request tags (Tag)s, and a fairness controller (FRCRT).

   FR, FRTag and Tags are used to guarantee the fairness of service among request messages. If a specific request message has not be served for a long time, the system stores the message in FR just beside the memory so that it has infinitely many chances to be served. This is necessary because it is possible that the message is in somewhere else whenever a memory is ready to serve a specific message. FRTag and Tags are used to control FR. When a cache generates a request message, the cache stores the address of the message in Tags. (See P9, P23, P30, P32 in Table 4.1 and VC2, VC5, VC10 in Table 4.2). s FRTag receives an address among Tags of caches in fair way and stores the address and the identifier of the cache. If FR is empty and a memory receive a request message whose address and source identifier match with FRTag, then the memory engine places the message into FR. The address in Tags and FRTag is

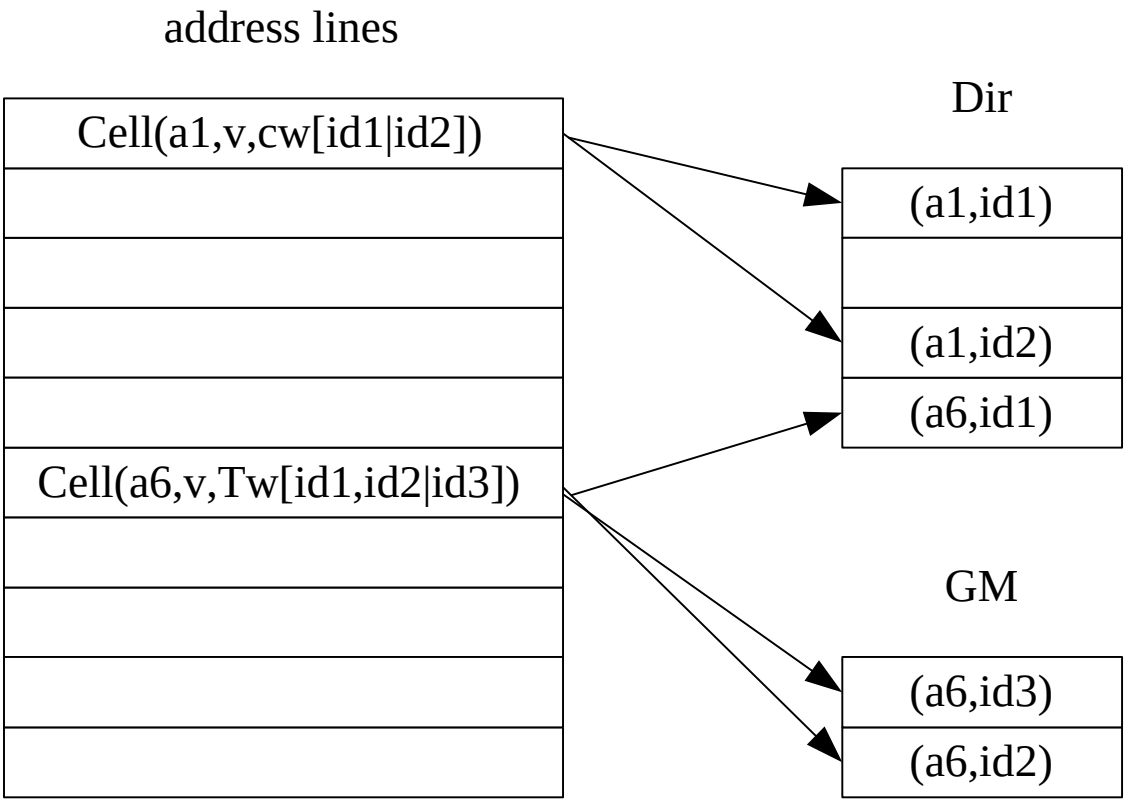

Figure 4-6: Shared Directory and Suspended Message Buffer

erased if the request message of FRTag is served by a memory. To control FRTag, we additionally use ErFRTag message that request the memory to erase FRTag.

FRCRT is used to guarantee the fairness of low priority message passing among the cache sites. If FRCRT is empty, the memory stores an identifier of a site in FRCRT to indicate the site who will send low priority message in the next time.

FR, FRTag, Tags, and FRCRT are used to avoid livelock that may occur with very low probability. To save the hardware cost, a system designer may not use these units and accept very low probability of livelock.

Table 4.1, Table 4.2, Table 4.3, Table 4.4, Table 4.5, Table 4.6, Table 4.7, and Table 4.8. show the specification of BCachet.

We use "$\Rightarrow$", "$\rightarrow$", "$\rightsquigarrow$", and "$\Rrightarrow$" to indicate that the message is using a high priority path, a low priority path, a stalled message queue, and the first priority request register, respectively. The notation "$SP(out)$" is the size of free space in output message queue of a memory, and $Full(gm)$ means that a buffer, gm, is full. SF besides a table means "strong fairness" of the rule. If a rule does not have SF, then it means "weak fairness" of the rule. In case of complicate rules, we use nested form of rules using "$Erase-(id, a)$ *Action*," "$Erase-id$ *Action*," and "$Stall-or-Nack\ Msg$." For example, WBB10 in Table 4.7 represents two rules (WBB10 with SNM3 and WBB10 with SNM4). WBB10 states that when the head of LIN is Wb, the HIN is empty, the memory state is $Cell(a, v_1, Tm[id, gm])$, and GM is full, then the memory engine can set the memory state as $Cell(a, v_1, Tw[\epsilon, \epsilon, gm])$ (WBB10) and can store Wb to STQ if STQ is not full (SNM3), or can negatively acknowledge Wb (SNM4).

Table 4.1: BCachet: The Processor Rules

Mandatory Processor Rules					
Instruction	**Cstate,Rqcnt,TAGs**	**Action**	**Next Cstate,Rqcnt,TAGs**		
$Loadl(a)$	$Cell(a, v, Clean_b)$	*retire*	$Cell(a, v, Clean_b)$	P1	SF
	$Cell(a, v, Dirty_b)$	*retire*	$Cell(a, v, Dirty_b)$	P2	SF
	$Cell(a, v, Clean_w)$	*retire*	$Cell(a, v, Clean_w)$	P3	SF
	$Cell(a, v, Dirty_w)$	*retire*	$Cell(a, v, Dirty_w)$	P4	SF
	$Cell(a, v, Clean_m)$	*retire*	$Cell(a, v, Clean_m)$	P5	SF
	$Cell(a, v, Dirty_m)$	*retire*	$Cell(a, v, Dirty_m)$	P6	SF
Continued on next page					

Table 4.1 – continued from previous page

Instruction	Cstate,Rqcnt,TAGs	Action	Next Cstate,Rqcnt,TAGs		
	$Cell(a, v, WbPending)$	*stall*	$Cell(a, v, WbPending)$	P7	
	$Cell(a, -, CachePending)$	*stall*	$Cell(a, -, CachePending)$	P8	
	$a \notin cache$, $rqcnt < rq_{max}, tags$	*stall*, $< CacheReq, a > \rightarrow H$	$Cell(a, -, CachePending)$, $rqcnt + 1, a \vert tags$	P9	SF
	$a \notin cache$, $rqcnt \geq rq_{max}, tags$	*stall*	$a \notin cache$, $rqcnt, a \vert tags$	P9A	
	$Cell(a, v, (Clean_b, Down_{wb}))$	*retire*	$Cell(a, v, (Clean_b, Down_{wb}))$	P10	SF
	$Cell(a, v, (Clean_b, Down_{mb}))$	*retire*	$Cell(a, v, (Clean_b, Down_{mb}))$	P11	SF
	$Cell(a, v, (Clean_w, Down_{mw}))$	*retire*	$Cell(a, v, (Clean_w, Down_{mw}))$	P12	SF
	$Cell(a, v, (Clean_w, \epsilon))$	*retire*	$Cell(a, v, (Clean_w, \epsilon))$	P13	SF
	$Cell(a, v, (Clean_m, \epsilon))$	*retire*	$Cell(a, v, (Clean_m, \epsilon))$	P14	SF
$Storel(a, v)$	$Cell(a, v, Clean_b)$	*retire*	$Cell(a, v, Dirty_b)$	P15	SF
	$Cell(a, v, Dirty_b)$	*retire*	$Cell(a, v, Dirty_b)$	P16	SF
	$Cell(a, v, Clean_w)$	*retire*	$Cell(a, v, Dirty_w)$	P17	SF
	$Cell(a, v, Dirty_w)$	*retire*	$Cell(a, v, Dirty_w)$	P18	SF
	$Cell(a, v, Clean_m)$	*retire*	$Cell(a, v, Dirty_m)$	P19	SF
	$Cell(a, v, Dirty_m)$	*retire*	$Cell(a, v, Dirty_m)$	P20	SF
	$Cell(a, v_1, WbPending)$	*stall*	$Cell(a, v_1, WbPending)$	P21	
	$Cell(a, -, CachePending)$	*stall*	$Cell(a, -, CachePending)$	P22	
	$a \notin cache$, $rqcnt < rq_{max}, tags$	*stall* $< CacheReq, a > \rightarrow H$	$Cell(a, -, CachePending)$, $rqcnt + 1, a \vert tags$	P23	SF
	$a \notin cache$, $rqcnt \geq rq_{max}, tags$	*stall*	$a \notin cache$, $rqcnt, tags$	P23A	SF
	$Cell(a, v, (Clean_b, Down_{wb}))$	*stall*	$Cell(a, v, (Clean_b, Down_{wb}))$	P24	
	$Cell(a, v, (Clean_b, Down_{mb}))$	*stall*	$Cell(a, v, (Clean_b, Down_{mb}))$	P25	
	$Cell(a, v, (Clean_w, Down_{mw}))$	*stall*	$Cell(a, v, (Clean_w, Down_{mw}))$	P26	
	$Cell(a, v, (Clean_w, \epsilon))$	*stall*	$Cell(a, v, (Clean_w, \epsilon))$	P27	
	$Cell(a, v, (Clean_m, \epsilon))$	*stall*	$Cell(a, v, (Clean_m, \epsilon))$	P28	
$Commit(a)$	$Cell(a, v, Clean_b)$	*retire*	$Cell(a, v, Clean_b)$	P29	SF
	$Cell(a, v, Dirty_b)$, $rqcnt < rq_{max}, tags$	*stall*, $< Wb, a, v > \rightarrow H$	$Cell(a, v, WbPending)$, $rqcnt + 1, a \vert tags$	P30	SF
	$Cell(a, v, Dirty_b)$, $rqcnt < rq_{max}, tags$	*stall*,	$Cell(a, v, Dirty_b)$, $rqcnt, tags$	P30A	SF
	$Cell(a, v, Clean_w)$	*retire*	$Cell(a, v, Clean_w)$	P31	SF
	$Cell(a, v, Dirty_w)$, $rqcnt < rq_{max}, tags$	*stall*, $< Wb, a, v > \rightarrow H$	$Cell(a, v, WbPending)$ $rqcnt + 1, a \vert tags$	P32	SF
	$Cell(a, v, Dirty_w)$, $rqcnt \geq rq_{max}, tags$	*stall*	$Cell(a, v, Dirty_w)$, $rqcnt, tags$	P32A	
	$Cell(a, v, Clean_m)$	*retire*	$Cell(a, v, Clean_m)$	P33	SF
	$Cell(a, v, Dirty_m)$	*retire*	$Cell(a, v, Dirty_m)$	P34	SF
	$Cell(a, v, WbPending)$	*stall*	$Cell(a, v, WbPending)$	P35	
	$Cell(a, -, CachePending)$	*stall*	$Cell(a, -, CachePending)$	P36	

Continued on next page

Table 4.1 – continued from previous page

Instruction	Cstate,Rqcnt,TAGs	Action	Next Cstate,Rqcnt,TAGs		
	$a \notin cache$	*retire*	$a \notin cache$	P37	SF
	$Cell(a, v, (Clean_b, Down_{wb}))$	*retire*	$Cell(a, v, (Clean_b, Down_{wb}))$	P38	SF
	$Cell(a, v, (Clean_b, Down_{mb}))$	*retire*	$Cell(a, v, (Clean_b, Down_{mb}))$	P39	SF
	$Cell(a, v, (Clean_w, \epsilon))$	*retire*	$Cell(a, v, (Clean_w, \epsilon))$	P40	SF
	$Cell(a, v, (Clean_w, Down_{mw}))$	*retire*	$Cell(a, v, (Clean_w, Down_{mw}))$	P41	SF
	$Cell(a, v, (Clean_m, \epsilon))$	*retire*	$Cell(a, v, (Clean_m, \epsilon))$	P42	SF
$Reconcile(a)$	$Cell(a, -, Clean_b)$	*retire*	$a \notin cache$	P43	SF
	$Cell(a, v, Dirty_b)$	*retire*	$Cell(a, v, Dirty_b)$	P44	SF
	$Cell(a, v, Clean_w)$	*retire*	$Cell(a, v, Clean_w)$	P45	SF
	$Cell(a, v, Dirty_w)$	*retire*	$Cell(a, v, Dirty_w)$	P46	SF
	$Cell(a, v, Clean_m)$	*retire*	$Cell(a, v, Clean_m)$	P47	SF
	$Cell(a, v, Dirty_m)$	*retire*	$Cell(a, v, Dirty_m)$	P48	SF
	$Cell(a, v, WbPending)$	*stall*	$Cell(a, v, WbPending)$	P49	
	$Cell(a, -, CachePending)$	*stall*	$Cell(a, -, CachePending)$	P50	
	$a \notin cache$	*retire*	$a \notin cache$	P51	SF
	$Cell(a, v, (Clean_b, Down_{wb}))$	*stall*	$Cell(a, v, (Clean_b, Down_{wb}))$	P52	
	$Cell(a, v, (Clean_b, Down_{mb}))$	*stall*	$Cell(a, v, (Clean_b, Down_{mb}))$	P53	
	$Cell(a, v, (Clean_w, \epsilon))$	*retire*	$Cell(a, v, (Clean_w, \epsilon))$	P54	SF
	$Cell(a, v, (Clean_w, Down_{mw}))$	*retire*	$Cell(a, v, (Clean_w, Down_{mw}))$	P55	SF
	$Cell(a, v, (Clean_m, \epsilon))$	*retire*	$Cell(a, v, (Clean_m, \epsilon))$	P56	SF

Table 4.2: BCachet: The Voluntary Cache Engine Rules

Voluntary C-engine Rules			
Cstate,Rqcnt,Hout,Tags	**Action**	**Next Cstate,Rqcnt,Tags**	
$Cell(a, -, Clean_b)$		$a \notin cache$	VC1
$Cell(a, v, Dirty_b)$, $rqcnt < rq_{max}, , tags$	$< Wb, a, v > \rightarrow H$	$Cell(a, v, WbPending)$, $rqcnt + 1, a \vert tags$	VC2
$Cell(a, v, Clean_w)$ $hout = \epsilon$	$< Down_{wb}, a > \Rightarrow H$	$Cell(a, v, Clean_b)$	VC3
$Cell(a, v, Dirty_w), hout = \epsilon$	$< Down_{wb}, a > \Rightarrow H$	$Cell(a, v, Dirty_b)$	VC4
$Cell(a, v, Dirty_w)$, $, rqcnt < rq_{max}, tags$	$< Wb, a, v > \rightarrow H$	$Cell(a, v, WbPending)$, $, rqcnt + 1, a \vert tags$	VC5
$Cell(a, v, Clean_m)$	$< Down_{mw}, a > \Rightarrow H$	$Cell(a, v, Clean_w)$	VC6
$hout = \epsilon$	$< Down_{mb}, a > \Rightarrow H$	$Cell(a, v, Clean_b)$	VC7
$Cell(a, v, Dirty_m)$	$< DownV_{mw}, a, v > \Rightarrow H$	$Cell(a, v, Clean_w)$	VC8
$hout = \epsilon$	$< DownV_{mb}, a, v > \Rightarrow H$	$Cell(a, v, Clean_b)$	VC9
$a \notin cache$, $rqcnt < rq_{max}, tags$	$< CacheReq, a > \rightarrow H$	$Cell(a, -, CachePending)$, $rqcnt + 1, a \vert tags$	VC10
$Cell(a, v, (Clean_w, \epsilon))$		$Cell(a, v, (Clean_b, Down_{wb}))$	VC11
$Cell(a, v, (Clean_w, Down_{mw}))$		$Cell(a, v, (Clean_b, Down_{mb}))$	VC12

Continued on next page

Table 4.2 – continued from previous page

	Cstate,Rqcnt,Hout,Tags	Action	Next Cstate,Rqcnt,Tags	
	$Cell(a, v, (Clean_m, \epsilon))$		$Cell(a, v, (Clean_w, Down_{mw}))$	VC13
			$Cell(a, v, (Clean_b, Down_{mb}))$	VC14

Table 4.3: BCachet: The Mandatory Cache Engine Rules

Mandatory C-engine Rules				
Msg from H	**Cstate,Hout,Tags**	**Action**	**Next Cstate,Rqcnt,Tags**	
$< Cache_w, a, v >$	$Cell(a, -, Clean_b)$		$Cell(a, v, Clean_w)$	MC1
	$Cell(a, v_1, Dirty_b)$		$Cell(a, v_1, Dirty_w)$	MC2
	$Cell(a, v_1, WbPending)$		$Cell(a, v_1, WbPending)$	MC3
	$Cell(a, -, CachePending)$		$Cell(a, v, (Clean_w, \epsilon))$	MC4
	$a \notin cache$		$Cell(a, v, Clean_w)$	MC5
$< Cache_m, a, v >$	$Cell(a, -, Clean_b)$		$Cell(a, v, Clean_m)$	MC6
	$Cell(a, v_1, Dirty_b)$		$Cell(a, v_1, Dirty_m)$	MC7
	$Cell(a, v_1, WbPending)$		$Cell(a, v_1, WbPending)$	MC8
	$Cell(a, -, CachePending)$		$Cell(a, v, (Clean_m, \epsilon))$	MC9
	$a \notin cache$		$Cell(a, v, Clean_m)$	MC10
$< Up_{wm}, a >$	$Cell(a, v, Clean_b)$		$Cell(a, v, Clean_b)$	MC11
	$Cell(a, v, Dirty_b)$		$Cell(a, v, Dirty_b)$	MC12
	$Cell(a, v, Clean_w)$		$Cell(a, v, Clean_m)$	MC13
	$Cell(a, v, Dirty_w)$		$Cell(a, v, Dirty_m)$	MC14
	$Cell(a, v, WbPending)$		$Cell(a, v, WbPending)$	MC15
	$Cell(a, -, CachePending)$		$Cell(a, -, CachePending)$	MC16
	$a \notin cache$		$a \notin cache$	MC17
	$Cell(a, v, (Clean_w, \epsilon))$		$Cell(a, v, (Clean_m, \epsilon)$	MC18
	$Cell(a, v, (Clean_b, Down_{wb}))$		$Cell(a, v, (Clean_b, Down_{wb})$	MC19
$< WbAck_b, a >$	$Cell(a, v, WbPending), rqcnt$ $hout = \epsilon, a \vert tags$	$< ErFRTag, a > \Rightarrow H$	$Cell(a, v, Clean_b), rqcnt - 1$ $tags$	MC20SF
$< DownReq_{wb}, a >$	$Cell(a, v, Clean_b)$		$Cell(a, v, Clean_b)$	MC21
	$Cell(a, v, Dirty_b)$		$Cell(a, v, Dirty_b)$	MC22
	$Cell(a, v, Clean_w), hout = \epsilon$	$< Down_{wb}, a > \Rightarrow H$	$Cell(a, v, Clean_b)$	MC23SF
	$Cell(a, v, Dirty_w), hout = \epsilon$	$< Down_{wb}, a > \Rightarrow H$	$Cell(a, v, Dirty_b)$	MC24SF
	$Cell(a, v, WbPending)$		$Cell(a, v, WbPending)$	MC25
	$Cell(a, -, CachePending)$		$Cell(a, -, CachePending)$	MC26
	$a \notin cache$		$a \notin cache$	MC27
	$Cell(a, v, (Clean_b, Down_{wb}))$		$Cell(a, v, (Clean_b, Down_{wb}))$	MC28
	$Cell(a, v, (Clean_b, Down_{mb}))$		$Cell(a, v, (Clean_b, Down_{mb}))$	MC29
	$Cell(a, v, (Clean_w, \epsilon))$		$Cell(a, v, (Clean_b, Down_{wb}))$	MC30
	$Cell(a, v, (Clean_w, Down_{mw}))$		$Cell(a, v, (Clean_b, Down_{mb}))$	MC31
$< DownReq_{mw}, a >$	$Cell(a, v, Clean_b)$		$Cell(a, v, Clean_b)$	MC32
	$Cell(a, v, Dirty_b)$		$Cell(a, v, Dirty_b)$	MC33

Continued on next page

Table 4.3 – continued from previous page

Msg from H	Cstate,Hout,Tags	Action	Next Cstate,Rqcnt,Tags	
	$Cell(a, v, Clean_w)$		$Cell(a, v, Clean_w)$	MC34
	$Cell(a, v, Dirty_w)$		$Cell(a, v, Dirty_w)$	MC35
	$Cell(a, v, Clean_m), hout = \epsilon$	$< Down_{mw}, a > \Rightarrow H$	$Cell(a, v, Clean_w)$	MC36SF
	$Cell(a, v, Dirty_m), hout = \epsilon$	$< DownV_{mw}, a, v > \Rightarrow H$	$Cell(a, v, Clean_w)$	MC37SF
	$Cell(a, v, WbPending)$		$Cell(a, v, WbPending)$	MC38
	$Cell(a, -, CachePending)$		$Cell(a, -, CachePending)$	MC39
	$a \notin cache$		$a \notin cache$	MC40
	$Cell(a, v, (Clean_b, Down_{wb}))$		$Cell(a, v, (Clean_b, Down_{wb}))$	MC41
	$Cell(a, v, (Clean_b, Down_{mb}))$		$Cell(a, v, (Clean_b, Down_{mb}))$	MC42
	$Cell(a, v, (Clean_w, \epsilon))$		$Cell(a, v, (Clean_w, \epsilon))$	MC43
	$Cell(a, v, (Clean_w, Down_{mw}))$		$Cell(a, v, (Clean_w, Down_{mw}))$	MC44
	$Cell(a, v, (Clean_m, \epsilon)$		$Cell(a, v, (Clean_w, Down_{mw}))$	MC45
$< DownReq_{mb}, a >$	$Cell(a, v, Clean_b)$		$Cell(a, v, Clean_b)$	MC46
	$Cell(a, v, Dirty_b)$		$Cell(a, v, Dirty_b)$	MC47
	$Cell(a, v, Clean_w), hout = \epsilon$	$< Down_{wb}, a > \Rightarrow H$	$Cell(a, v, Clean_b)$	MC48SF
	$Cell(a, v, Dirty_w), hout = \epsilon$	$< Down_{wb}, a > \Rightarrow H$	$Cell(a, v, Dirty_b)$	MC49SF
	$Cell(a, v, Clean_m), hout = \epsilon$	$< Down_{mb}, a > \Rightarrow H$	$Cell(a, v, Clean_b)$	MC50SF
	$Cell(a, v, Dirty_m), hout = \epsilon$	$< DownV_{mb}, a, v > \Rightarrow H$	$Cell(a, v, Clean_b)$	MC51SF
	$Cell(a, v, WbPending)$		$Cell(a, v, WbPending)$	MC52
	$Cell(a, -, CachePending)$		$Cell(a, -, CachePending)$	MC53
	$a \notin cache$		$a \notin cache$	MC54
	$Cell(a, v, (Clean_b, Down_{wb}))$		$Cell(a, v, (Clean_b, Down_{wb}))$	MC55
	$Cell(a, v, (Clean_b, Down_{mb})$		$Cell(a, v, (Clean_b, Down_{mb}))$	MC56
	$Cell(a, v, (Clean_w, \epsilon))$		$Cell(a, v, (Clean_b, Down_{wb}))$	MC57
	$Cell(a, v, (Clean_w, Down_{mw}))$		$Cell(a, v, (Clean_b, Down_{mb}))$	MC58
	$Cell(a, v, (Clean_m, \epsilon))$		$Cell(a, v, (Clean_b, Down_{mb}))$	MC59
$< CacheAck, a, v >$	$Cell(a, -, CachePending)$	$< ErFRTag, a > \Rightarrow H$	$Cell(a, v, Clean_b)$	MC60SF
	$rqcnt, hout = \epsilon, a \vert tags$		$rqcnt - 1, tags$	
$< CacheAck, a >$	$Cell(a, v, (Clean_b, Down_{wb}))$,	$< Down_{wb}, a > \Rightarrow H$	$Cell(a, v, Clean_b)$	MC61SF
	$rqcnt, hout = \epsilon, a \vert tags$	$< ErFRTag, a > \Rightarrow H$	$rqcnt - 1, tags$	
	$Cell(a, v, (Clean_b, Down_{mb}))$,	$< Down_{mb}, a > \Rightarrow H$	$Cell(a, v, Clean_b)$	MC62SF
	$rqcnt, hout = \epsilon$	$< ErFRTag, a > \Rightarrow H$	$rqcnt - 1, tags$	
	$Cell(a, v, (Clean_w, \epsilon))$,	$< ErFRTag, a > \Rightarrow H$	$Cell(a, v, Clean_w)$	MC63SF
	$rqcnt, hout = \epsilon, a \vert tags$		$rqcnt - 1, tags$	
	$Cell(a, v, (Clean_w, Down_{mw}))$	$< Down_{mw}, a > \Rightarrow H$	$Cell(a, v, Clean_w))$	MC64SF
	$rqcnt, hout = \epsilon, a \vert tags$	$< ErFRTag, a > \Rightarrow H$	$rqcnt - 1, tags$	
	$Cell(a, v, (Clean_m, \epsilon))$	$< ErFRTag, a > \Rightarrow H$	$Cell(a, v, Clean_m)$	MC65SF
	$rqcnt, hout = \epsilon, a \vert tags$		$rqcnt - 1, tags$	
$< CacheNack, a >$	$Cell(a, v, CachePending)$	$< CacheReq, a > \rightarrow H$	$Cell(a, v, CachePending)$	MC66SF
$< WbNack, a >$	$Cell(a, v, WbPending)$	$< Wb, a, v > \rightarrow H$	$Cell(a, v, WbPending)$	MC67SF

Table 4.4: BCachet: The Voluntary Memory Engine Rules

Voluntary M-engine Rules				
LIN,OUT	**Mstate**	**Action**	**Next Mstate**	
$lin = \epsilon, SP(out) \geq 2$	$Cell(a, v, Cw[dir])$	$< Cache_w, a, v > \Rightarrow id$	$Cell(a, v, Cw[id\|dir])$	VM1
	$Cell(a, v, Cw[id])$	$< Up_{wm}, a > \Rightarrow id$	$Cell(a, v, Cm[id])$	VM2
	$Cell(a, v, Cw[\epsilon])$	$< Cache_m, a, v > \Rightarrow id$	$Cell(a, v, Cm[id])$	VM3
	$Cell(a, v, Cw[id\|dir])$	$< DownReq_{wb}, a > \Rightarrow id$	$Cell(a, v, Tw[id, dir, \epsilon])$	VM4
	$Cell(a, v, Cm[id])$	$< DownReq_{mw}, a > \Rightarrow id$	$Cell(a, v, T'm[id])$	VM5
		$< DownReq_{mb}, a > \Rightarrow id$	$Cell(a, v, Tm[id, \epsilon])$	VM6
	$Cell(a, v, T'm[id])$	$< DownReq_{wb}, a > \Rightarrow id$	$Cell(a, v, Tm[id, \epsilon])$	VM7

Table 4.5: BCachet: The Memory Engine Rules-Hmsg

Mandatory M-engine Rules				
Hin	**Mstate**	**Action**	**Next Mstate**	
$< Down_{wb}, a, id >$	$Cell(a, v, Cw[id\|dir])$		$Cell(a, v, Cw[dir])$	HM1
	$Cell(a, v, Tw[id\|dir_1, dir_2, gm])$		$Cell(a, v, Tw[dir_1, dir_2, gm])$	HM2A
	$Cell(a, v, Tw[dir_1, id\|dir_2, gm])$		$Cell(a, v, Tw[dir_1, dir_2, gm])$	HM2B
	$Cell(a, v, Cm[id])$		$Cell(a, v, Cw[\epsilon])$	HM3
	$Cell(a, v, T'm[id])$		$Cell(a, v, Cw[\epsilon])$	HM4
	$Cell(a, v, Tm[id, gm])$		$Cell(a, v, Tw[\epsilon, \epsilon, gm])$	HM5
$< Down_{mw}, a, id >$	$Cell(a, v, Cm[id])$		$Cell(a, v, Cw[id])$	HM6
	$Cell(a, v, T'm[id])$		$Cell(a, v, Cw[id])$	HM7
	$Cell(a, v, Tm[id, gm])$		$Cell(a, v, Tw[id, \epsilon, gm])$	HM8
$< DownV_{mw}, a, v, id >$	$Cell(a, v_1, Cm[id])$		$Cell(a, v, Cw[id])$	HM9
	$Cell(a, v_1, T'm[id])$		$Cell(a, v, Cw[id])$	HM10
	$Cell(a, v_1, Tm[id, \epsilon])$		$Cell(a, v, Tw[id, \epsilon, \epsilon])$	HM11A
	$Cell(a, v_1, Tm[id, gm])(gm \neq \epsilon)$		$Cell(a, v_1, Tw[id, \epsilon, gm])$	HM11B
$< Down_{mb}, a, id >$	$Cell(a, v, Cm[id])$		$Cell(a, v, Cw[\epsilon])$	HM12
	$Cell(a, v, T'm[id])$		$Cell(a, v, Cw[\epsilon])$	HM13
	$Cell(a, v, Tm[id, gm])$		$Cell(a, v, Tw[\epsilon, \epsilon, gm])$	HM14
$< DownV_{mb}, a, v, id >$	$Cell(a, v_1, Cm[id])$		$Cell(a, v, Cw[\epsilon])$	HM15
	$Cell(a, v_1, T'm[id])$		$Cell(a, v, Cw[\epsilon])$	HM16
	$Cell(a, v_1, Tm[id, \epsilon])$		$Cell(a, v, Tw[\epsilon, \epsilon, \epsilon])$	HM17A
	$Cell(a, v_1, Tm[id, gm])(gm \neq \epsilon)$		$Cell(a, v_1, Tw[\epsilon, \epsilon, gm])$	HM17B
$< ErFRtag, a, id >$		$Erase - (id, a)$ *Action*		HM18
Out	**Mstate,FRTag**	**Action**	**Next Mstate,FRTag**	
$SP(Out) \geq 2$	$Cell(a, v, Tw[\epsilon, \epsilon, id\|gm])$ $frtag = (id, a)$	$< WbAck_b, a > \Rightarrow id$ $Erase - (id, a)$ *Action*	$Cell(a, v, Tw[\epsilon, \epsilon, gm])$ $frtag = \epsilon$	GM1 SF
	$Cell(a, v, Tw[\epsilon, \epsilon, \epsilon])$		$Cell(a, v, Cw[\epsilon])$	GM2

Continued on next page

Table 4.5 – continued from previous page

Hin	Mstate	Action	Next Mstate		
$SP(Out)\geq 2$	$Cell(a, v, Tw[dir_1, id\|dir_2, gm])$	$< DownReq_{wb}, a > \Rightarrow id$	$Cell(a, v, Tw[id\|dir_1, dir_2, gm])$	DM1	SF

Erase-(id,a) Action				
Id,Address	**FRTag**	**Action**	**Next FRTag**	
(id, a)	$frtag = (id, a)$		$frtag = \epsilon$	FRTEr1
	$frtag \neq (id, a)$		$frtag$	FRTEr2

Table 4.6: BCachet: The Memory Engine Rules-FR

Mandatory M-engine Rules					
FR	**Mstate,OUT**	**Action**	**Next Mstate**		
$< CacheReq, a, id >$	$Cell(a, v, Cw[dir])(id \notin dir, SP(out)\geq 2)$	1. $< CacheAck, a, v > \Rightarrow id$ $Erase-(id, a)$ $Action$	$Cell(a, v, Cw[dir])$	FRC1	SF
	$Cell(a, v, Cw[dir])(id \in dir, SP(out)\geq 2)$	$< CacheAck, a > \Rightarrow id$ $Erase-(id, a)$ $Action$	$Cell(a, v, Cw[dir])$	FRC2	SF
	$Cell(a, v, Tw[dir_1, dir_2, gm])(id \in dir_1 \| dir_2, SP(out)\geq 2)$	$< CacheAck, a > \Rightarrow id$ $Erase-(id, a)$ $Action$	$Cell(a, v, Tw[dir_1, dir_2, gm])$	FRC3	SF
	$Cell(a, v, Cm[id])(SP(out)\geq 2)$	$< CacheAck, a > \Rightarrow id$ $Erase-(id, a)$ $Action$	$Cell(a, v, Cm[id])$	FRC4	SF
	$Cell(a, v, T'm[id])(SP(out)\geq 2)$	$< CacheAck, a > \Rightarrow id$ $Erase-(id, a)$ $Action$	$Cell(a, v, T'm[id])$	FRC5	SF
	$Cell(a, v, Tm[id, gm])(SP(out)\geq 2)$	$< CacheAck, a > \Rightarrow id$ $Erase-(id, a)$ $Action$	$Cell(a, v, Tm[id, gm])$	FRC6	SF
	$Cell(a, v, Cm[id_1])(SP(out)\geq 2)$	$< DownReq_{mw}, a > \Rightarrow id$	$Cell(a, v, T'm[id_1])$	FRC7	SF
$< Wb, a, v, id >$,	$Cell(a, v_1, Cw[dir])$ $(id \notin dir, \neg Full(gm))$	$Erase-(id, a)$ $Action$	$Cell(a, v, Tw[\epsilon, dir, id])$	FRW1	SF
	$Cell(a, v_1, Tw[dir_1, dir_2, gm])(id \notin dir_1 \| dir_2, dir_1 \| dir_2 \neq \epsilon, \neg Full(gm))$	$Erase-(id, a)$ $Action$	$Cell(a, v, Tw[dir_1, dir_2, gm \| id])$	FRW2	SF
	$Cell(a, v_1, Cw[id\|dir])(\neg Full(gm))$	$Erase-(id, a)$ $Action$	$Cell(a, v, Tw[\epsilon, dir, id])$	FRW3	SF
	$Cell(a, v_1, Tw[id\|dir_1, dir_2, gm])(\neg Full(gm))$	$Erase-(id, a)$ $Action$	$Cell(a, v, Tw[dir_1, dir_2, gm \| id])$	FRW4A	SF
	$Cell(a, v_1, Tw[dir_1, id\|dir_2, gm])(\neg Full(gm))$	$Erase-(id, a)$ $Action$	$Cell(a, v, Tw[dir_1, dir_2, gm \| id])$	FRW4B	SF
	$Cell(a, v_1, Cm[id_1])$ $(id \neq id_1, SP(out)\geq 2, \neg Full(gm))$	$< DownReq_{mb}, a > \Rightarrow id_1$ $Erase-(id, a)$ $Action$	$Cell(a, v, Tm[id_1, id])$	FRW5	SF
	$Cell(a, v_1, T'm[id_1])$ $(id \neq id_1, SP(out)\geq 2, \neg Full(gm))$	$< DownReq_{wb}, a > \Rightarrow id_1$ $Erase-(id, a)$ $Action$	$Cell(a, v, Tm[id_1, id])$	FRW6	SF
	$Cell(a, v_1, Tm[id_1, gm])$ $(id \neq id_1, \neg Full(gm))$	$Erase-(id, a)$ $Action$	$Cell(a, v, Tm[id_1, gm \| id])$	FRW7	SF

Continued on next page

Table 4.6 – continued from previous page

FR	Mstate,OUT	Action	Next Mstate	
	$Cell(a, v_1, Cm[id])(\neg Full(gm))$	$Erase-(id,a)$ *Action*	$Cell(a, v, Tw[\epsilon, \epsilon, id])$	FRW8 SF
	$Cell(a, v_1, T'm[id])(\neg Full(gm))$	$Erase-(id,a)$ *Action*	$Cell(a, v, Tw[\epsilon, \epsilon, id])$	FRW9 SF
	$Cell(a, v_1, Tm[id, gm])(\neg Full(gm))$	$Erase-(id,a)$ *Action*	$Cell(a, v, Tw[\epsilon, \epsilon, gm\|id])$	FRW10 SF
	$Full(gm)$	$Erase-id$ *Action* *stall message in FR*		FRW11 SF

Erase-id Action				
Id	**Mstate**	**Action**	**Next Mstate**	
id	$Cell(a, v_1, Cw[id\|dir])$		$Cell(a, v_1, Tw[\epsilon, dir, \epsilon])$	WEr1
	$Cell(a, v_1, Tw[id\|dir_1, dir_2, gm])$		$Cell(a, v_1, Tw[dir_1, dir_2, gm])$	WEr2A
	$Cell(a, v_1, Tw[dir_1, id\|dir_2, gm])$		$Cell(a, v_1, Tw[dir_1, dir_2, gm])$	WEr2B
	$Cell(a, v_1, Cm[id])$		$Cell(a, v_1, Tw[\epsilon, \epsilon, \epsilon])$	WEr3
	$Cell(a, v_1, T'm[id])$		$Cell(a, v_1, Tw[\epsilon, \epsilon, \epsilon])$	WEr4
	$Cell(a, v_1, Tm[id, gm])$		$Cell(a, v_1, Tw[\epsilon, \epsilon, gm])$	WEr5
	otherwise	*no action*		WEr6

Erase-(id,a) Action				
Id,Address	**FRTag**	**Action**	**Next ErFRTag**	
(id, a)	$frtag = (id, a)$		$frtag = \epsilon$	FRTEr1
	$frtag \neq (id, a)$		$frtag$	FRTEr2

Table 4.7: BCachet: The Memory Engine Rules-LIN

Mandatory M-engine Rules				
Head(Lin),Hin	**Mstate,OUT**	**Action**	**Next Mstate**	
$< CacheReq, a, id >$, $hin = \epsilon$	$Cell(a, v, Cw[dir])$ $(id \notin dir, SP(out) \geq 2)$	1. $< CacheAck, a, v > \Rightarrow id$ $Erase-(id,a)$ *Action*	$Cell(a, v, Cw[dir])$	CHA1 SF
	$Cell(a, v, Tw[dir_1, dir_2, gm])$ $(id \notin dir_1 \| dir_2)$	$Stall-or-Nack$ *message*	$Cell(a, v, Tw[dir_1, dir_2, gm])$	CHA2 SF
	$Cell(a, v, Cw[dir])$ $(id \in dir, SP(out) \geq 2)$	$< CacheAck, a > \Rightarrow id$ $Erase-(id,a)$ *Action*	$Cell(a, v, Cw[dir])$	CHA3 SF
	$Cell(a, v, Tw[dir_1, dir_2, gm])(id \in dir_1 \| dir_2, SP(out) \geq 2)$	$< CacheAck, a > \Rightarrow id$ $Erase-(id,a)$ *Action*	$Cell(a, v, Tw[dir_1, dir_2, gm])$	CHA4 SF
	$Cell(a, v, Cm[id_1])$ $(id \neq id_1, SP(out) \geq 2)$	1.$Stall-or-Nack$ *message* 2. $< DownReq_{mw}, a > \Rightarrow id_1$	$Cell(a, v, T'm[id_1])$	CHA5 SF
	$Cell(a, v, T'm[id_1])$ $(id \neq id_1, \neg Full(stq))$	$Stall-or-Nack$ *message*	$Cell(a, v, T'm[id_1])$	CHA6 SF
	$Cell(a, v, Tm[id_1, gm])$	$Stall-or-Nack$ *message*	$Cell(a, v, Tm[id_1, gm])$	CHA7 SF

Continued on next page

Table 4.7 – continued from previous page

Head(Lin),Hin	Mstate,OUT	Action	Next Mstate		
	$(id \neq id_1)$				
	$Cell(a, v, Cm[id])$ $(SP(out) \geq 2)$	$< CacheAck, a > \Rightarrow id$ $Erase-(id, a)$ $Action$	$Cell(a, v, Cm[id])$	CHA8	SF
	$Cell(a, v, T'm[id])$ $(SP(out) \geq 2)$	$< CacheAck, a > \Rightarrow id$ $Erase-(id, a)$ $Action$	$Cell(a, v, T'm[id])$	CHA9	SF
	$Cell(a, v, Tm[id, gm])$ $(SP(out) \geq 2)$	$< CacheAck, a > \Rightarrow id$ $Erase-(id, a)$ $Action$	$Cell(a, v, Tm[id, gm])$	CHA10	SF
	FRTag,FR	**Action**	**Next FRTag**		
	$frtag = (id, a), fr = \epsilon$	$< CacheReq, a, id > \Rrightarrow fr$	$frtag = \epsilon$	CHB1	SF
$< Wb, a, v, id >,$ $hin = \epsilon$	$Cell(a, v_1, Cw[dir])(id \notin dir,$ $\neg Full(gm), SP(out) \geq 2)$		$Cell(a, v, Tw[\epsilon, dir, id])$	WBA1	SF
	$Cell(a, v_1, Tw[dir_1, dir_2,$ $gm])(id \notin dir_1 \| dir_2,$ $dir_1 \| dir_2 \neq \epsilon, \neg Full(gm))$		$Cell(a, v, Tw[dir_1, dir_2,$ $gm \| id])$	WBA2	SF
	$Cell(a, v_1, Cw[id \| dir])$ $(\neg Full(gm), SP(out) \geq 2)$		$Cell(a, v, Tw[\epsilon, dir, id])$	WBA3	SF
	$Cell(a, v_1, Tw[id \| dir_1, dir_2,$ $gm])(\neg Full(gm))$		$Cell(a, v, Tw[dir_1, dir_2,$ $gm \| id])$	WBA4A	SF
	$Cell(a, v_1, Tw[dir_1, id \| dir_2,$ $gm])(\neg Full(gm))$		$Cell(a, v, Tw[dir_1, dir_2,$ $gm \| id])$	WBA4B	SF
	$Cell(a, v_1, Cm[id_1])(id \neq id_1,$ $\neg Full(gm), SP(out) \geq 2)$	$< DownReq_{mb}, a > \Rightarrow id_1$	$Cell(a, v, Tm[id_1, id])$	WBA5	SF
	$Cell(a, v_1, T'm[id_1])(id \neq id_1,$ $\neg Full(gm), SP(out) \geq 2)$	$< DownReq_{wb}, a > \Rightarrow id_1$	$Cell(a, v, Tm[id_1, id])$	WBA6	SF
	$Cell(a, v_1, Tm[id_1, gm])$ $(id \neq id_1, \neg Full(gm))$		$Cell(a, v, Tm[id_1, gm \| id])$	WBA7	SF
	$Cell(a, v_1, Cm[id])$ $(\neg Full(gm), SP(out) \geq 2)$		$Cell(a, v, Tw[\epsilon, \epsilon, id])$	WBA8	SF
	$Cell(a, v_1, T'm[id])$ $(\neg Full(gm))$		$Cell(a, v, Tw[\epsilon, \epsilon, id])$	WBA9	SF
	$Cell(a, v_1, Tm[id, gm])$ $(\neg Full(gm))$		$Cell(a, v, Tw[\epsilon, \epsilon, gm \| id])$	WBA10	SF
	$Cell(a, v_1, Cw[id_1 \| dir])$ $(id \notin dir, id \neq id_1,$ $Full(gm), SP(out) \geq 2)$	$< DownReq_{wb}, a > \Rightarrow id_1$ $Stall-or-Nack\ message$	$Cell(a, v_1, Tw[id_1, dir, \epsilon])$	WBB1	SF
	$Cell(a, v_1, Tw[dir_1, dir_2,$ $gm])(id \notin dir_1 \| dir_2,$ $Full(gm))$	$Stall-or-Nack\ message$	$Cell(a, v_1, Tw[dir_1, dir_2,$ $gm])$	WBB2	SF
	$Cell(a, v_1, Cw[id \| dir])$ $(Full(gm))$	$Stall-or-Nack\ message$	$Cell(a, v_1, Tw[\epsilon, dir, \epsilon])$	WBB3	SF
	$Cell(a, v_1, Tw[id \| dir_1, dir_2$ $, gm])(Full(gm))$	$Stall-or-Nack\ message$	$Cell(a, v_1, Tw[dir_1, dir_2,$ $gm])$	WBB4A	SF

Continued on next page

Table 4.7 – continued from previous page

Head(Lin),Hin	Mstate,OUT	Action	Next Mstate	
	$Cell(a, v_1, Tw[dir_1, id \mid dir_2, gm])(Full(gm))$	$Stall-or-Nack\ message$	$Cell(a, v_1, Tw[dir_1, dir_2, gm])$	WBB4B SF
	$Cell(a, v_1, Cm[id_1])$ $(id \neq id_1,$ $Full(gm), SP(out) \geq 2)$	$< DownReq_{mb}, a > \Rightarrow id_1$ $Stall-or-Nack\ message$	$Cell(a, v_1, Tm[id_1, \epsilon])$	WBB5 SF
	$Cell(a, v_1, T'm[id_1])(id \neq id_1,$ $Full(gm), SP(out) \geq 2)$	$< DownReq_{wb}, a > \Rightarrow id_1$ $Stall-or-Nack\ message$	$Cell(a, v_1, Tm[id_1, \epsilon])$	WBB6 SF
	$Cell(a, v_1, Tm[id_1, gm])$ $(id \neq id_1, Full(gm))$	$Stall-or-Nack\ message$	$Cell(a, v_1, Tm[id_1, gm])$	WBB7 SF
	$Cell(a, v_1, Cm[id])$ $(Full(gm)$	$Stall-or-Nack\ message$	$Cell(a, v_1, Tw[\epsilon, \epsilon, \epsilon])$	WBB8 SF
	$Cell(a, v_1, T'm[id])$ $(Full(gm))$	$Stall-or-Nack\ message$	$Cell(a, v_1, Tw[\epsilon, \epsilon, \epsilon])$	WBB9 SF
	$Cell(a, v_1, Tm[id, gm])$ $(Full(gm))$	$Stall-or-Nack\ message$	$Cell(a, v_1, Tw[\epsilon, \epsilon, gm])$	WBB10 SF
	FRTag,FR	**Action**	**Next FRTag**	
	$frtag = (id, a), fr = \epsilon$	$< Wb, a, v, id > \Rrightarrow fr$	$frtag = \epsilon$	WBC1 SF

Erase-(id,a) Action				
Id,Address	**FRTag**	**Action**	**Next FRTag**	
(id, a)	$frtag = (id, a)$		$frtag = \epsilon$	FRTEr1
	$frtag \neq (id, a)$		$frtag$	FRTEr2

Stall-or-Nack Msg				
Msg	**STQ,OUT**	**Action**		
$< CacheReq, a, id >$	$\neg Full(stq)$	$< CacheReq, a, id > \rightsquigarrow stq$		SNM1
	$Full(stq), SP(out) \geq 2$	$< CacheNack, a > \Rightarrow id$		SNM2
$< Wb, a, v, id >$	$\neg Full(stq)$	$< Wb, a, v, id > \rightsquigarrow stq$		SNM3
	$Full(stq), SP(out) \geq 2$	$< WbNack, a > \Rightarrow id$		SNM4

Table 4.8: BCachet: The Mandatory Message Passing Rules

Mandatory Message Passing Rules		
	$Sys(Msite(mem, hin, lin, msg; stq, fr, out, frtag, frcrt), sites),\ if \neg Full(lin)$	STQ SF
$\rightarrow$	$Sys(Msite(mem, hin, lin; msg, stq, fr, out, frtag, frcrt), sites)$	
	$Sys(Msite(mem, \epsilon, lin, stq, fr, out, frtag, frcrt),$ $Site(id, cache, rqcnt, tags, in, msg; fifo, lout, pmb, mpb, proc) \mid sites)$	HPS SF
$\rightarrow$	$Sys(Msite(mem, msg, lin, stq, fr, out, frtag, frcrt),$ $Site(id, cache, rqcnt, tags, in, fifo, lout, pmb, mpb, proc) \mid sites)$	
	$Sys(Msite(mem, hin, lin, stq, fr, out, frtag, id),$ $Site(id, cache, rqcnt, tags, in, \epsilon, msg; lout, pmb, mpb, proc) \mid sites)$	LPS SF
	Continued on next page	

Table 4.8 – continued from previous page

Mandatory Message Passing Rules		
	$if \neg Full(lin)$	
$\rightarrow$	$Sys(Msite(mem, hin, lin; msg, stq, fr, out, frtag, \epsilon),$	
	$Site(id, cache, rqcnt, tags, in, \epsilon, lout, pmb, mpb, proc) \mid sites)$	
	$Sys(Msite(mem, hin, lin, stq, fr, msg; out, frtag, frcrt),$	
	$Site(id, cache, rqcnt, tags, in, hout, lout, pmb, mpb, proc) \mid sites)$	MPS
	$if \neg Full(in)$	
$\rightarrow$	$Sys(Msite(mem, hin, lin, stq, fr, out, frtag, frcrt),$	
	$Site(id, cache, rqcnt, tags, in; msg, hout, lout, pmb, mpb, proc) \mid sites)$	
	$Sys(Msite(mem, hin, lin, stq, fr, out, frtag, \epsilon),$	
	$Site(id, cache, rqcnt, tags, in, hout, lout, pmb, mpb, proc) \mid sites)$	FRCSF
	$if\ lout \neq \epsilon$	
$\rightarrow$	$Sys(Msite(mem, hin, lin, stq, fr, out, frtag, id),$	
	$Site(id, cache, rqcnt, tags, in, hout, lout, pmb, mpb, proc) \mid sites)$	
	$Sys(Msite(mem, hin, lin, stq, fr, out, \epsilon, frcrt),$	
	$Site(id, cache, rqcnt, a \mid tags, in, hout, lout, pmb, mpb, proc) \mid sites)$	FRTSF
$\rightarrow$	$Sys(Msite(mem, hin, lin, stq, fr, out, (id, a), frcrt),$	
	$Site(id, cache, rqcnt, a \mid tags, in, hout, lout, pmb, mpb, proc) \mid sites)$	

Chapter 5

Correctness of BCachet

In this chapter, we show a brief proof of the soundness of BCachet and offer arguments of the liveness of BCachet. The soundness of BCachet states that any state transition in BCachet can be simulated in HWb, and consequently, in CRF. The liveness of BCachet states that any memory instruction in processor-to-memory buffer will eventually be served. We do not give a detail proof of the liveness of BCachet because it is too tedious and too long and is not very insightful.

Notations In this chapter, we will use notations in terms of a BCachet term s for simple description as follows.

$s = Sys(Msite(mem, hin, lin, stq, fr, out, frtag, frcrt),$
$\quad Site(id, cache, rqcnt, tags, in, hout, lout, pmb, mbp, proc)|sites)$

$$\begin{aligned} Mem(s) &\equiv mem \\ Hin(s) &\equiv hin \\ Lin(s) &\equiv lin \\ STQ(s) &\equiv stq \\ FR(s) &\equiv fr \\ Out(s) &\equiv out \end{aligned}$$

$$
\begin{aligned}
FRTag(s) &\equiv frtag \\
FRCRT(s) &\equiv frcrt \\
Cache_{id}(s) &\equiv cache \\
Rqcnt_{id}(s) &\equiv rqcnt \\
Tags_{id}(s) &\equiv tags \\
In_{id}(s) &\equiv in \\
Hout_{id}(s) &\equiv hout \\
Lout_{id}(s) &\equiv lout \\
Pmb_{id}(s) &\equiv pmb \\
Mpb_{id}(s) &\equiv mpb \\
Proc_{id}(s) &\equiv proc
\end{aligned}
$$

Using the definitions above, we also define composite FIFOs as follows.

$$
\begin{aligned}
FRLinSTQLout_{id}(s) &= FR(s); Lin(s); STQ(s); Lout_{id}(s) \\
LinSTQLout_{id}(s) &= Lin(s); STQ(s); Lout_{id}(s) \\
HinHout_{id}(s) &= Hin(s); Hout(s) \\
In_{id}Out(s) &= In_{id}(s); Out(s)
\end{aligned}
$$

To describe, cache states and memory states, we define notations as follows.

$$
\begin{aligned}
Locked(cs) \equiv\ & cs = (Clean_b, Down_{wb}) \vee cs = (Clean_b, Down_{mb}) \vee cs = (Clean_w, \epsilon) \vee \\
& cs = (Clean_w, Down_{mw}) \vee cs = (Clean_m, \epsilon)
\end{aligned}
$$

$$
Dir(ms) \equiv \begin{cases}
dir & if\ ms = Cw[dir] \\
dir_1|dir_2 & if\ ms = Tw[dir_1, dir_2, -, -] \\
id & if\ ms = Cm[id] \vee ms = Tm[id, -] \vee ms = T'm[id, -]
\end{cases}
$$

$$GM(ms) \equiv \begin{cases} gm & if\ ms = Tw[-,-,-,gm] \vee ms = Tm[-,gm] \\ \epsilon & if\ ms = Cw[-] \vee ms = T'm[-] \vee ms = Cm[-] \end{cases}$$

$$GM(ms) - id \equiv \begin{cases} gm & if\ GM(ms) = id|gm \\ GM(ms) & if\ id \notin GM(ms) \end{cases}$$

5.1 Soundness of BCachet

In this section, we show simple proof of the soundness of BCachet. The soundness of BCachet states that for a given state transition "$s_1 \twoheadrightarrow s_2$" in BCachet, there exists a mapping function $f(BCachet \mapsto HWb)$, such that, $f(s_1) \twoheadrightarrow f(s_2)$. We prove the soundness of BCachet in two steps. First, we define a mapping function f. Second, we show that each rewriting step in BCachet corresponds to a rewriting step or a sequence of rewriting steps in HWb under the mapping, f. Therefore, any state transition of BCachet can be simulated by HWb.

5.1.1 Mapping from BCachet to HWb

Invariants in BCachet Before we start defining a mapping function, we first state lemmas about request messages. The following tree lemmas can be proved by induction on rewriting steps.

Lemma 1. *For a given BCachet term s,*

$$\begin{aligned} & In_{id}Out(s) = fifo_1; Msg(H, id, WbNack, a, -); fifo_2 \\ \Longrightarrow\ & Msg(id, H, Wb, a, -) \notin FRLinSTQLout_{id}(s)\ \wedge \\ & Msg(H, id, WbAck_b, a, -) \notin In_{id}Out(s)\ \wedge \\ & Msg(H, id, WbNack, a, -) \notin fifo_1; fifo_2\ \wedge\ Cell(a, v_1, Tw[-,-,-,id|-]) \notin Mem(s)\ \wedge \\ & Cell(a, -, Tm[-,id|-]) \notin Mem(s)\ \wedge\ Cell(a, -, WbPending) \in Cache_{id}(s) \end{aligned}$$

Lemma 2. *For a given BCachet term s,*

$$
\begin{aligned}
& In_{id}Out(s) = fifo_1; Msg(H, id, CacheNack, a, -); fifo_2 \\
\Longrightarrow\ & Msg(id, H, CacheReq, a, -) \notin FRLinSTQLout_{id}(s)\ \wedge \\
& Msg(H, id, CacheAck, a, -) \notin In_{id}Out(s)\ \wedge \\
& Msg(H, id, CacheNack, a, -) \notin fifo_1; fifo_2\ \wedge \\
& Cell(a, v, CachePending) \in Cache_{id}(s)
\end{aligned}
$$

Definition 1 (CacheReq-Related Message). *CacheReq, CacheAck, and CacheNack are CacheReq-related message type.*

Definition 2 (CacheReq-Related Message). *Wb, $WbAck_b$, WbNack, and id in the suspended message buffer are Wb-related message type.*

Definition 3 (Reqeust-Related Message). *CacheReq, Wb, $WbAck_b$, CacheAck, CacheNack, WbNack, and id in the suspended message buffer are request-related message type.*

Lemma 3. *In a given BCachet term s, at maximum, there can exist one request-related message for the same address and regarding the same site in the term s, that is, for a given BCachet term s, only one of (5.1), (5.2), and (5.3) can be true at a time as well as (5.4), (5.5), and (5.6) are always true.*

$$msg \in FRLinSTQLout_{id}(s),\ such\ that,\ Addr(msg) = a \wedge Src(msg) = id \tag{5.1}$$

$$
\begin{aligned}
& msg \in In_{id}Out(s),\ such\ that, \\
& (Cmd(msg) = CacheAck \vee Cmd(msg) = WbAck_b \vee Cmd(msg) = CacheNack \\
& \vee Cmd(msg) = WbNack) \wedge Addr(msg) = a \wedge Dest(msg) = id
\end{aligned} \tag{5.2}
$$

$$Cell(a, v_1, ms) \in Mem(s),\ such\ that,\ id \in GM(ms) \tag{5.3}$$

$$
\begin{aligned}
& FRLinSTQLout_{id}(s) = fifo_1; msg; fifo_2 \; such \; that \\
& Addr(msg) = a \wedge Src(msg) = id \\
\Longrightarrow \quad & \forall msg_1 \in fifo_1; fifo_2, (Addr(msg_1) \neq a) \vee (Src(msg_1) \neq id)
\end{aligned} \tag{5.4}
$$

$$
\begin{aligned}
& In_{id}Out(s) = fifo_1; msg; fifo_2 \; such \; that \\
& (Cmd(msg) = CacheAck \vee Cmd(msg) = WbAck_b \vee Cmd(msg) = CacheNack \vee \\
& Cmd(msg) = WbNack) \; \wedge Addr(msg) = a \wedge Dest(msg) = id \\
\Longrightarrow \quad & msg_1 \notin fifo_1; fifo_2 \; such \; that \\
& (Cmd(msg_1) = CacheAck \vee Cmd(msg_1) = WbAck_b \vee Cmd(msg_1) = CacheNack \vee \\
& Cmd(msg_1) = WbNack) \wedge Addr(msg_1) = a \wedge Dest(msg_1) = id
\end{aligned} \tag{5.5}
$$

$$
id \in GM(ms) \Longrightarrow id \notin GM(ms) - id \tag{5.6}
$$

Lemma 3 states there cannot be more than two request-related messages for the same address and for same *id*. Lemma 1 and Lemma 2 state that if a WbNack or a CacheNack message is in the memory-to-cache path, then the target cache block of the message must be in WbPending or CachePending state, respectively. These lemmas will be used later in proving that the mapping function f is well defined.

We define a mapping function that maps a BCachet term to a HWb term to prove the soundness of BCachet. We define the mapping part by part. First, we define several sub-mapping functions that map a sub-term of a BCachet term to a sub-term of a HWb term because the mapping is too complicate. These functions are defined in the notion of converting rules. Each sub-mapping function is related to a set of converting rules which are strongly terminating and confluent. Using sub-mapping functions, we finally define a mapping function that maps a BCachet term to a HWb term. We do not use a singe set of converting rules to directly define a mapping function from BCachet to HWb because it is difficult to make the rules confluent and it is not easy to express a mapped HWb term in terms of a BCachet term using a single set of rules.

First, we define a function (cv_{Er}::$FIFO \rightarrow FIFO$) that erases all WbNacks, CacheNacks, ErFRTags, and dummy CacheAcks in a FIFO. We define converting rules associated with cv_{Er} as follows.

Erase-ErFRTag-and-Nacks Rule

$$
\begin{array}{ll}
 & fifo_1; msg; fifo_2 \\
 & if\ \ Cmd(msg) = ErFRTag\ \vee\ Cmd(msg) = WbNack\ \vee\ Cmd(msg) = CacheNack \\
\rightarrow & fifo_1; fifo_2
\end{array}
$$

Erase-CacheAck Rule

$$
\begin{array}{ll}
 & fifo_1; Msg(src, dest, CacheAck, a, value); fifo_2 \\
 & if\ value = \epsilon \\
\rightarrow & fifo_1; fifo_2
\end{array}
$$

Convert-CacheAck Rule

$$
\begin{array}{ll}
 & fifo_1; Msg(src, dest, CacheAck, a, value); fifo_2 \\
 & if\ value \neq \epsilon \\
\rightarrow & fifo_1; Msg(src, dest, Cache_b, a, value); fifo_2
\end{array}
$$

Definition 4 (Erase-Dummy,Convert-CacheAck Rules).

$$
\begin{array}{lll}
Cv_{Er} & \equiv & \{Erase - ErFRTag - and - Nacks\ Rule,\ Erase - CacheAck\ Rule, \\
 & & Convert - CacheAck\ Rule\}
\end{array}
$$

Cv_{Er} is a set of rules that erase dummy messages or convert non-dummy CacheAck message. If CacheAck has a value, it is not a dummy message, otherwise a dummy message. There is no corresponding message of dummy messages in HWb. Therefore, these messages can be safely erased. Non-CacheAck message corresponds to $Cache_b$ in HWb, and thus, it is converted to $Cache_b$.

Lemma 4. *Cv_{Er} is strongly terminating and confluent, that is, rewriting a FIFO term with respect to Cv_{Er} always terminates and reaches the same normal form, regardless of the order in which the rules are applied.*

Proof. The proof of termination is trivial because according to the rules, the rules consume ErFRTag, CacheAck, WbNack, and CacheNack, and do not generate these types of messages. The confluence follows from the fact that the converting rules do not interfere with each other. □

We indicate the normal form of a FIFO term, s_{FIFO}, with respect to Cv_{Er} as $cv_{Er}(s_{FIFO})$. For example, for a given BCachet term s, $cv_{Er}(Out(s))$ is the output message queue of the memory site of s ($Out(s)$), with all of its dummy CacheAcks, CacheNacks, and WbNacks are erased and non-dummy CacheAcks are converted to Cache_b. For a given BCachet term s, we will use cv_{Er} later to convert $Out(s)$, $In_{id}(s)$, $Hin_{id}(s)$, and $Hout_{id}(s)$ to the corresponding units in HWb.

Second, we define a function cv_{Rqs}::$(FIFO, CACHE, PtoPFIFO) \rightarrow (FIFO, CACHE, PtoPFIFO)$ that converts a FIFO, a cache of a BCachet term, and a point-to-point buffer. We define converting rules associated with cv_{Rqs} as follows.

Convert-WbNack Rule

$$\begin{aligned} & (fifo_1; Msg(H, id, WbNack, a, -); fifo_2, Cell(a, v, WbPending) | cache, ptopfifo) \\ \rightarrow \quad & (fifo_1; fifo_2, Cell(a, v, WbPending) | cache, ptopfifo \odot Msg(id, H, Wb, a, v)) \end{aligned}$$

Convert-CacheNack Rule

$$\begin{aligned} & (fifo_1; Msg(H, id, CacheNack, a, -); fifo_2, Cell(a, -, CachePending) | cache, ptopfifo) \\ \rightarrow \quad & (fifo_1; fifo_2, Cell(a, -, CachePending) | cache, ptopfifo \odot Msg(id, H, CacheReq, a, -)) \end{aligned}$$

Definition 5 (Convert Nacks Rules).

$$Cv_{Rqs} \equiv \{Convert - WbNack\ Rule,\ Convert - CacheNack\ Rule\}$$

Lemma 5. *For a given BCachet term s, Cv_{Rqs} is strongly terminating and confluent with respect to $(In_{id}Out(s),\ Cache_{id}(s),\ \epsilon)$, that is, rewriting process starting from $(In_{id}Out(s),\ Cache_{id}(s),\ \epsilon)$ always terminates and reaches the same normal form, regardless of the order in which the rules are applied.*

Proof. The proof of termination is trivial because according to the rules, the rules consume CacheNack and WbNack. The confluence follows from the fact that the converting rules do not interfere with each other. This is because two Nack messages that have the same address and the same destination cannot coexist in $In_{id}Out(s)$ by Lemma 3. □

For a given BCachet term s, we notate the normal form of $(In_{id}Out(s), Cache_{id}(s), \epsilon)$ with respect to Cv_{Rqs} as $cv_{Rqs}(In_{id}Out(s), Cache_{id}(s), \epsilon)$. We define $rqs_{id}(s)$ as the third element of $cv_{Rqs}(In_{id}Out(s), Cache_{id}(s), \epsilon)$. The formal expression of $rqs_{id}(s)$ is as follows.

$$\begin{aligned} rqs_{id}(s) &= Third(cv_{Rqs}(In_{id}Out(s), Cache_{id}(s), \epsilon)) \\ Third(x, y, z) &= z \end{aligned}$$

For a given BCachet term s, all CacheNacks and WbNacks for site id in a BCachet term s are converted to CacheReqs and Wbs and stored in $rqs_{id}(s)$ because the corresponding Pending block must be in $Cache_{id}(s)$ by Lemma 1 and Lemma 2.

Third, we define a function cv_{Rls}::$(CACHE, PtoPFIFO) \rightarrow (CACHE, PtoPFIFO)$ that converts BCachet' cache and a point-to-point buffer to HWb's cache and a point-to-point buffer. To define cv_{Rls}, we define converting rules associated with it as follows.

Release-Message-of-$(Clean_b, Down_{wb})$ Rule

$$\begin{aligned} & (Cell(a, v, (Clean_b, Down_{wb}))|cache, ptopfifo) \\ \rightarrow\ & (Cell(a, v, Clean_b)|cache, ptopfifo \odot Msg(id, H, Down_{wb}, a, -)) \end{aligned}$$

Release-Message-of-$(Clean_b, Down_{mb})$ Rule

$$\begin{aligned} & (Cell(a, v, (Clean_b, Down_{mb}))|cache, ptopfifo) \\ \rightarrow\ & (Cell(a, v, Clean_b)|cache, ptopfifo \odot Msg(id, H, Down_{mb}, a, -)) \end{aligned}$$

Release-Message-of-$(Clean_w, \epsilon)$ Rule

$$\begin{aligned} & (Cell(a, v, (Clean_w, \epsilon))|cache, ptopfifo) \\ \rightarrow \quad & (Cell(a, v, Clean_w)|cache, ptopfifo) \end{aligned}$$

Release-Message-of-$(Clean_w, Down_{mw})$ Rule

$$\begin{aligned} & (Cell(a, v, (Clean_w, Down_{mw}))|cache, ptopfifo) \\ \rightarrow \quad & (Cell(a, v, Clean_w)|cache, ptopfifo \odot Msg(id, H, Down_{mw}, a, -)) \end{aligned}$$

Release-Message-of-$(Clean_m, \epsilon)$ Rule

$$\begin{aligned} & (Cell(a, v, (Clean_m, \epsilon))|cache, ptopfifo) \\ \rightarrow \quad & (Cell(a, v, Clean_m)|cache, ptopfifo) \end{aligned}$$

Definition 6 (Release Message Rules).

$$\begin{aligned} Cv_{Rls} \quad \equiv \quad & \{Release - Message - of - (Clean_b, Down_{wb})Rule, \\ & Release - Message - of - (Clean_b, Down_{mb})Rule, \\ & Release - Message - of - (Clean_w, \epsilon)Rule, \\ & Release - Message - of - (Clean_w, Down_{mw})Rule, \\ & Release - Message - of - (Clean_m, \epsilon)Rule\} \end{aligned}$$

Lemma 6. *For a given BCachet term s, Cv_{Rls} is strongly terminating and confluent with respect to $(Cache_{id}(s), \epsilon)$.*

Proof. The proof of termination is trivial because according to the rules, the rules consume $(Clean_b, Down_{wb})$, $(Clean_b, Down_{mb})$, $(Clean_w, \epsilon)$, $(Clean_w, Down_{mw})$, and $(Clean_m, \epsilon)$, and no rule can generate these types of states. The confluence follows from the fact that two different cells cannot have the same address and the converting rules do not interfere with each other. □

For a given BCachet term s, we notate the normal form of $(Cache_{id}(s), \epsilon)$ with respect to Cv_{Rls} as $cv_{Rls}(Cache_{id}(s), \epsilon)$. We define $rcache{::}CACHE \rightarrow CACHE$ and $rmsgs{::}CACHE$

$\rightarrow PtoPFIFO$ as the first and the second element of $cv_{Rls}(Cache_{id}(s), \epsilon)$. The formal expression of these are as follows.

$$\begin{aligned} rcache(cache) &= First(cv_{rls}(cache, \epsilon))) \\ rmsgs(cache) &= Second(cv_{rls}(cache, \epsilon))) \\ First(x, y) &= x \\ Second(x, y) &= y \end{aligned}$$

For a given BCachet term s, $rcache(Cache_{id}(s))$ and $rmsgs(Cache_{id}(s))$ are the cache and the output message queue of the cache site (id) of the corresponding HWb term, respectively. $rcache(Cache_{id}(s))$ is achieved by releasing messages held in expanded cache states. $rmsgs(Cache_{id}(s))$ is a point-to-point buffer that contains messages released by $Cache_{id}(s)$.

Fourth, we define $cv_{fifo} :: FIFO \rightarrow PtoPFIFO$ that converts a FIFO to a point-to-point buffer. We define Convert-FIFO Rule and also define Cv_{fifo} as a set of this rule. For a given FIFO term, s_{fifo}, $cv_{fifo}(s_{fifo})$ is the normal form of s_{fifo} with respect to Cv_{fifo}.

Convert-FIFO Rule

$$msg_1; msg_2 \rightarrow msg_1 \odot msg_2$$

Definition 7 (Convert FIFO Rules).

$$Cv_{fifo} \equiv \{Convert - FIFORule\}$$

Lemma 7. *For a given FIFO term s_{fifo}, Cv_{fifo} is strongly terminating and confluent with respect to s_{fifo}.*

Proof. The proof of termination is trivial because according to the rules, they consume ";". No rule can generate ";". The confluence follows from the fact that converting rule do not interfere with itself. □

Using the functions defined above, we define a mapping function f that maps a BCachet term to a HWb term. For a given BCachet term s that has n sites ($id_1 \sim id_n$), we define $f(s)$ as follows.

$$
\begin{aligned}
f(s) &= Sys(msite, site_{id_1}|site_{id_2}|...site_{id_n}) \\
msite &= Msite(Mem(s), cv_{fifo}(cv_{Er}(Hin(s)))\odot cv_{fifo}(FR(s))\odot cv_{fifo}(Lin(s))\odot \\
&\quad cv_{fifo}(STQ(s))\odot Ms_{id_1}(s)\odot Ms_{id_2}(s)\odot ...Ms_{id_n}(s), cv_{fifo}(cv_{Er}(out))) \\
Ms_{id_k}(s) &= cv_{fifo}(cv_{Er}(Hout_{id_k}(s)))\odot cv_{fifo}(Lout_{id_k}(s))\odot rqs_{id_k}(s) \ \ (1{\leq}k{\leq}n) \\
site_{id_k} &= Site(id_k, rcache(Cache_{id_k}(s))), cv_{fifo}(cv_{Er}(In_{id_k}(s))), rmsgs(Cache_{id_k}(s)), \\
&\quad Pmb_{id_k}(s), Mpb_{id_k}(s), Proc_{id_k}(s)) \ \ (1{\leq}k{\leq}n)
\end{aligned}
$$

The following theorem states that BCachet is sound with respect to HWb, that is, any state transition in BCachet can be simulated by a sequence of state transitions in HWb.

Theorem 1 (HWb Simulate BCachet). *Given BCachet terms s_1 and s_2,*
$s_1 \overset{BCachet}{\twoheadrightarrow} s_2$ *in BCachet* $\Rightarrow f(s_1) \overset{HWb}{\twoheadrightarrow} f(s_2)$ *in HWb*

Proof. For a given state transition in BCachet, $s_1 \overset{r_{BCachet}}{\rightarrow} s_2$, we can easily find the corresponding HWb rule (r_{HWb}) such that $f(s_1) \overset{r_{HWb}}{\twoheadrightarrow} f(s_2)$. The list of $r_{BCachet}$ and r_{HWb} are as shown in Table 5.1, Table 5.2, Table 5-1, Table 5.3, Table 5.4, Table 5.5, Table 5.6, and Table 5.7. Starting from an initial BCachet term where all queues and caches are empty, we can show the theorem by induction. □

$r_{BCachet}$	r_{HWb}
P1, P10, P11	P1
P2	P2
P3, P12, P13	P3
P4	P4
P5, P14	P5
P6	P6
P7	P7
P8	P8
P9	P9+Message-Cache-to-Mem
P9A	ϵ
P15	P10
P16	P11
P17	P12
P18	P13
P19	P14
P20	P15
P21	P16
P22	P17
P23	P18+Message-Cache-to-Mem
P23A, P24, P25, P26, P27, P28	ϵ
P29, P38, P39	P19
P30	P20+Message-Cache-to-Mem
P31, P40, P41	P21
P32	P22+Message-Cache-to-Mem
P33, P42	P23
P34	P24
P35	P25
P36	P26
P37	P27
P30A, P32A	ϵ
P43	P28
P44	P29
P45, P54, P55	P30
P46	P31
P47, P56	P32
P48	P33
P49	P34
P50	P35
P51	P36
P52, P53	ϵ

Table 5.1: Simulation of BCachet Processor Rules in Table 4.1 by HWb Processor Rules in Table 3.1

$r_{BCachet}$	r_{HWb}
VC1	C1
VC2	C2+Message-Cache-to-Mem
VC3	C3+Message-Cache-to-Mem
VC4	C4+Message-Cache-to-Mem
VC5	C5+Message-Cache-to-Mem
VC6	C6+Message-Cache-to-Mem
VC7	C7+Message-Cache-to-Mem
VC8	C8+Message-Cache-to-Mem
VC9	C9+Message-Cache-to-Mem
VC10	C10+Message-Cache-to-Mem
VC11, VC12	C3
VC13	C6
VC14	C7

Table 5.2: Simulation of BCachet Voluntary Cache Engine Rules in Table 4.2 by HWb Voluntary Cache Engine Rules in Table 3.2

$r_{BCachet}$	r_{HWb}
MC1	C12
MC2	C13
MC3	C14
MC4	C15
MC5	C16
MC6	C17
MC7	C18
MC8	C19
MC9	C20
MC10	C21
MC11, MC19	C22
MC12	C23
MC13, MC18	C24
MC14	C25
MC15	C26
MC16	C27
MC17	C28
MC20	C29
MC21, MC28, MC29	C30
MC22	C31
MC23	C32+Message-Cache-to-Mem
MC24	C33+Message-Cache-to-Mem
MC25	C34
MC26	C35
MC27	C36
MC30, MC31	C32
MC32, MC41, MC42	C37
MC33	C38
MC34, MC43, MC44	C39
MC35	C40
MC36	C41+Message-Cache-to-Mem
MC37	C42+Message-Cache-to-Mem
MC38	C43
MC39	C44
MC40	C45
MC45	C41
MC46, MC55, MC56	C46
MC47	C47
MC48	C48+Message-Cache-to-Mem
MC49	C49+Message-Cache-to-Mem
MC50	C50+Message-Cache-to-Mem
MC51	C51+Message-Cache-to-Mem
MC52	C52
MC53	C53
MC54	C54
MC57, MC58	C48
MC59	C50
MC60	C11
MC61, MC62, MC63, MC64, MC65	ϵ
MC66	ϵ
MC67	ϵ

Figure 5-1: Simulation of BCachet Mandatory Cache Engine Rules in Table 4.3 by HWb Mandatory Cache Engine Rules in Table 3.2

$r_{BCachet}$	r_{HWb}
VM1	M1
VM2	M2
VM3	M3
VM4	M4
VM5	M5
VM6	M6
VM7	M7

Table 5.3: Simulation of BCachet Voluntary Memory Engine Rules in Table 4.4 by HWb Voluntary Memory Engine Rules in Table 3.3

$r_{BCachet}$	r_{HWb}
HM1	M1
HM2A	M2A
HM2B	M2B
HM3	M3
HM4	M4
HM5	M5
HM6	M6
HM7	M7
HM8	M8
HM9	M9
HM10	M10
HM11A	M11A
HM11B	M11B
HM12	M12
HM13	M13
HM14	M14
HM15	M15
HM16	M16
HM17A	M17A
HM17B	M17B
HM18	ϵ
GM1	M18
GM2	M19
DM1	MDir1

Table 5.4: Simulation of BCachet Mandatory Memory Engine Rules-Hmsg in Table 4.5 by HWb Mandatory Memory Engine Rules-B in Table 3.4

$r_{BCachet}$	r_{HWb}
FRC1	M8
FRC2	M10
FRC3	M11
FRC4	M15
FRC5	M16
FRC6	M17
FRC7	M12
FRW1	M18
FRW2	M19A
FRW3	M20A
FRW4A	M21A1
FRW4B	M21A2
FRW5	M22
FRW6	M23
FRW7	M24
FRW8	M25A
FRW9	M26A
FRW10	M27A
FRW11 with WEr1	M20B
FRW11 with WEr2A	M21B1
FRW11 with WEr2B	M21B2
FRW11 with WEr3	M25B
FRW11 with WEr4	M26B
FRW11 with WEr5	M27B
FRW11 with WEr6	ϵ

Table 5.5: Simulation of BCachet Mandatory Memory Engine Rules-FR in Table 4.6 by HWb Mandatory Memory Engine Rules-A in Table 3.3

$r_{BCachet}$	r_{HWb}
CHA1	M8
CHA2	M9
CHA3	M10
CHA4	M11
CHA5	M12
CHA6	M13
CHA7	M14
CHA8	M15
CHA9	M16
CHA10	M17
CHB1	ϵ
WBA1	M18
WBA2	M19A
WBA3	M20A
WBA4A	M21A1
WBA4B	M21A2
WBA5	M22
WBA6	M23
WBA7	M24
WBA8	M25A
WBA9	M26A
WBA10	M27A
WBB1	M4
WBB2	ϵ
WBB3	M20B
WBB4A	M21B1
WBB4B	M21B2
WBB5	M6
WBB6	M7
WBB7	ϵ
WBB8	M25B
WBB9	M26B
WBB10	M27C
WBC1	ϵ

Table 5.6: Simulation of BCachet Mandatory Memory Engine Rules-LIN in Table 4.7 by HWb Mandatory Memory Engine Rules-A in Table 3.3

$r_{BCachet}$	r_{HWb}
STQ, FRC, FRT	ϵ
HPS, LPS	Message-Cache-to-Mem
MPS	Message-Mem-to-Cache

Table 5.7: Simulation of BCachet Mandatory Message Passing Rules in Table 4.8 by Message Passing Rules in HWb

5.2 Liveness of BCachet

In this section, we give some arguments about the liveness of BCachet. We will not show detail proof because the proof is too tedious and too long.

Liveness of BCachet: For a given BCachet sequence $\sigma < s_1, s_2, ... >$,

$$
\begin{aligned}
< t, Loadl(-) > \in Pmb_{id}(\sigma) &\quad \leadsto \quad < t, - > \in Mpb_{id}(\sigma) \\
< t, Storel(-) > \in Pmb_{id}(\sigma) &\quad \leadsto \quad < t, Ack > \in Mpb_{id}(\sigma) \\
< t, Commit(-) > \in Pmb_{id}(\sigma) &\quad \leadsto \quad < t, Ack > \in Mpb_{id}(\sigma) \\
< t, Reconcile(-) > \in Pmb_{id}(\sigma) &\quad \leadsto \quad < t, Ack > \in Mpb_{id}(\sigma)
\end{aligned}
$$

The liveness of BCachet states that every memory instruction in a processor-to-memory buffer (Pmb) will eventually be executed.

The liveness of BCachet can be proved in five steps. The brief arguments about the five proving steps are as follows:

1. We can prove the liveness of message flow, that is, any message will eventually reach its destination.

   Liveness of Message Flow: For a given BCachet sequence $\sigma < s_1, s_2, ... >$,

   $$
   \begin{aligned}
   msg \in HinHout_{id}(\sigma) &\quad \leadsto \quad Head(HinHout_{id}(\sigma)) = msg \\
   msg \in In_{id}Out(\sigma) \wedge Dest(msg) = id &\quad \leadsto \quad Head(In_{id}Out(\sigma)) = msg \\
   msg \in LinSTQLout_{id}(\sigma) &\quad \leadsto \quad Head(LinSTQLout_{id}(\sigma)) = msg
   \end{aligned}
   $$

   It is trivial that H-path is live, that is, any message will eventually be sunk at the memory. This is because these messages are guaranteed to be sunk at the memory in any condition. (See rules HM1-HM18 in Table 4.5).

   We can prove the liveness of memory-to-cache path based on the liveness of the H-path and the fact that the head message of memory-to-cache path can be sunk, in the worst situation, if the HOUT or LOUT of a cache have empty slots for new messages. (See

rules MC20, MC23, MC24, MC36, MC37, MC48-MC51, MC60-MC67 in Figure 4.3). The memory-to-cache path is live because HOUT will eventually have an empty slots and LOUT always have a space for new messages due to the size constraint of LOUT, "$Size(LOUT) \geq rq_{max}$."

We can prove that L-path is live, that is, any request type messages in L-path can arrive at its destination memory. The rules consuming the head message of this path can be fired, in the worst situation, if the outgoing message queue (OUT) of the memory has more than two empty slots. OUT will eventually have more than two empty slots because memory-to-cache path is live, and thus, OUT will eventually send its messages.

2. We can prove the liveness of downgrading, that is, any transient memory state will eventually become stable state.

Liveness of Downgrading: For a given BCachet sequence $\sigma <s_1, s_2, ...>$,

$$\begin{aligned} Cell(a, -, Tw[-, -, -, gm]) \in Mem(\sigma) &\leadsto Cell(a, -, Tw[\epsilon, \epsilon, \epsilon, gm|-]) \in Mem(\sigma) \\ &\leadsto Cell(a, -, Cw[\epsilon] \in Mem(\sigma) \end{aligned}$$

$$\begin{aligned} Cell(a, -, Tm[id, gm]) \in Mem(\sigma) &\leadsto Cell(a, -, Tw[\epsilon, \epsilon, \epsilon, gm|-]) \in Mem(\sigma) \quad (5.7) \\ &\leadsto Cell(a, -, Cw[\epsilon]) \in Mem(\sigma) \end{aligned}$$

$$\begin{aligned} Cell(a, -, T'm[id]) \in Mem(\sigma) &\leadsto Cell(a, -, Cw[\epsilon]) \in Mem(\sigma) \vee \\ &\quad Cell(a, -, Cw[id]) \in Mem(\sigma) \end{aligned}$$

Trivially, a memory cell in Tw state will eventually send all DownReq messages kept in the second directory because the rules sending DownReq are strongly fair and the message flow is live. (See rules DM1 in Table 4.5). The response of DownReq (Down) may be held in cache states if the cache generated a CacheReq and it is not completed. The cache states holding Down messages are $(Clean_w, Down_{mw})$, $(Clean_b, Down_{wb})$, and $(Clean_b, Down_{mb})$, and in this case, the memory contains id of the CacheReq in its

directory. If the memory contains the *id* of the CacheReq, then the CacheReq can be completed at the memory and the memory can send *CacheAck* to the cache. (See rules FRC3, FRC5, FRC6 in Table 4.6 and CHA4, CHA9, CHA10 in Table 4.7). Therefore, the Down messages held in caches will eventually be released and the memory in Tw state will eventually receive all requested Down messages. After receiving all responses to DownReqs, the directory of the memory becomes empty. Once, the memory in Tw state has empty directory, it is easy to show Tw state will eventually becomes Cw state. (See rules GM1, GM2 in Table 4.5).

Using the same argument, we can prove that a memory block in Tm state will eventually receive the response to down request sent by the memory block and become Tw state. Therefore, Tm state will eventually become stable state because Tw state will eventually become stable state.

Using the same argument, we can prove that a memory block in T'm state also will eventually receive the response to the down request sent by the memory block and become Cw state.

3. We can prove the liveness of first priority request control, that is, the first priority request register (FR) and the first priority request register's tag (FRTag) will eventually become empty as well as a message in FR will eventually be served and a request message will enter FR if it cannot be served forever.

Liveness of First Priority Request Control: For a given BCachet sequence σ $<s_1, s_2, ...>$,

$$FR(\sigma)\neq\epsilon \quad \leadsto \quad FR(\sigma) = \epsilon$$

$$FR(\sigma) = Msg(id, H, Wb, a, -) \quad \leadsto \quad FR(\sigma) = \epsilon \wedge Cell(a, -, ms) \in Mem(\sigma) : id \in GM(ms)$$

$$FR(\sigma) = Msg(id, H, CacheReq, a, -) \quad \leadsto \quad FR(\sigma) = \epsilon \wedge Msg(H, id, CacheAck, a, -) \in Out(\sigma)$$

$$
\begin{aligned}
FRTag(\sigma) = (id, a) \quad \leadsto \quad & FRTag(\sigma) = \epsilon \\
& \Box(Msg(id, H, Wb, a, -) \in LinSTOLout_{id}(\sigma) \vee \\
& Msg(H, id, WbNack, a, -) \in In_{id}Out(\sigma)) \\
\leadsto \quad & FR(\sigma) = Msg(id, H, Wb, a, -)
\end{aligned}
$$

$$
\begin{aligned}
& \Box(Msg(id, H, CacheReq, a, -) \in LinSTOLout_{id}(\sigma) \vee \\
& Msg(H, id, CacheNack, a, -) \in In_{id}Out(\sigma)) \\
\leadsto \quad & FR(\sigma) = Msg(id, H, CacheReq, a, -)
\end{aligned}
$$

We can prove that the message in FR will eventually be served using the liveness of downgrading. If a CacheReq is stored in FR, the memory has sent or will eventually send necessary DownReq messages, and the memory state is or will eventually become a transient state. (See FRC7 in Table 4.6). The stalled CacheReq eventually be served because the transient state will eventually be a stable state that can serve the CacheReq and the rule that can serve the CacheReq is strongly fair (See rules FRC1-FRC6 in Table 4.6). If a Wb is stored in FR, the major reason why the Wb cannot be served is that there are not enough space in the suspended message buffer. The Wb will eventually be served because the liveness of downgrading guarantees that all transient state will become stable state, and the suspended message buffer earns an empty slot for the Wb message during the state transition toward a stable state.

FRTag will eventually become empty so that another request message can reserve the FR by listing its address and *id* in FRTag. This is trivial because when a request in FR is served, then the memory engine erases the address and identifier kept in FRTag. (See rules FRC1-FRC6, FRW1-FRW10 in Table 4.6, MC20, MC60-MC65 in Table 4.3, and HM18 in Table 4.5).

If a request message cannot be served all the time, then the message eventually enter FR so that it will eventually be served. This follows from the strong fairness of the rule, FRT, in Table 4.8.

4. We can prove that request is live, that is, any pending state (CachePending, WbPending, and locked states) will eventually become a clean state.

Liveness Proof of Request: For a given BCachet sequence $\sigma <s_1, s_2, ...>$,

$$
\begin{aligned}
& Cell(a, -, cs) \in Cache_{id}(\sigma) : Locked(cs) = True \vee cs = CachePending \\
\rightsquigarrow \quad & Cell(a, -, Clean_b) \in Cache_{id}(\sigma) \vee Cell(a, -, Clean_w) \in Cache_{id}(\sigma) \vee \\
& Cell(a, -, Clean_m) \in Cache_{id}(\sigma)
\end{aligned}
$$

$$
\begin{aligned}
& Cell(a, -, WbPending) \in Cache_{id}(\sigma) \\
\rightsquigarrow \quad & Cell(a, -, Clean_b) \in Cache_{id}(\sigma)
\end{aligned}
$$

The liveness of request is based on and the liveness of first priority request control. A request message will eventually enter FR if a request message is not served all the times. It is guaranteed that the message in FR will eventually be served, and therefore, all pending state will eventually become stable state.

5. BCachet is live, that is, all instructions in a processor-to-memory buffer of a cache site will eventually be served and the corresponding responses will eventually be placed in the memory-to-processor buffer of the site.

Liveness of BCachet: For a given BCachet sequence $\sigma <s_1, s_2, ...>$,

$$
\begin{aligned}
< t, Loadl(-) > \in Pmb_{id}(\sigma) \quad & \rightsquigarrow \quad < t, - > \in Mpb_{id}(\sigma) \\
< t, Storel(-) > \in Pmb_{id}(\sigma) \quad & \rightsquigarrow \quad < t, Ack > \in Mpb_{id}(\sigma) \\
< t, Commit(-) > \in Pmb_{id}(\sigma) \quad & \rightsquigarrow \quad < t, Ack > \in Mpb_{id}(\sigma) \\
< t, Reconcile(-) > \in Pmb_{id}(\sigma) \quad & \rightsquigarrow \quad < t, Ack > \in Mpb_{id}(\sigma)
\end{aligned}
$$

We can prove the liveness of BCachet based on the liveness of request. Since all pending state will eventually become stable state, it is easy to show that all memory instruction will eventually be served by the strong fairness of processor rules. (See Table 4.1).

Chapter 6

Conclusion

This thesis addresses a buffer management method for Cachet. We modified the Cachet protocol and changed the hardware structure for the protocol. The modifications are done in two phases.

In the first phase, we modified the protocol without changing the assumption of point-to-point message passing. In this phase, we split atomic coarse-grained writeback and multi-casting of DownReq operations into finer-grained actions so that the free buffer space required by these actions is small. We stores the value of a writeback message in the memory at the moment of reception of the writeback message without affecting the memory model. This modification allows the memory engine to store only the identifier of the writeback message in the suspended message buffer so that the suspended message buffer saves the storage for the value of the writeback message. We simplify the protocol in this phase too. We eliminate some redundancy and assign higher priorities on some rules.

In the second phase, we change the cache-to-memory message path from point-to-point buffer to a couple of high and low priority FIFOs so that we can use simple FIFOs in a real implementation. Due to the existence of voluntary rules in Cachet, separating message paths based on the message type causes an incorrect reordering problem. We solved this problem by message holding technique, in which expanded cache states store the information messages until sending message becomes safe. We also use fairness control units to prevent livelock. These units guarantee that a request will be served eventually by giving high priority to the request if the request suffers from unfair treatment. As a final design, the concrete buffer management for Cachet, BCachet is shown in Chapter 4.

In the remainder of this chapter, Section 6.1 discusses the advantage of the BCachet implementation. The comparison of the hardware costs of BCachet and Cachet is also shown in this section. In Section 6.2, we discuss possible future work related to this thesis.

6.1 Advantage of BCachet

To discuss the real scalability, we will discuss the merits of BCachet based on the assumption that the memory space is also distributed over the system.

In BCachet, the reordering ability of the incoming message buffer of the memory is greatly reduced from $P \times N$ FIFOs to two FIFOs where P is the number of processors and N is the number of memory blocks in a memory unit. We also reduced the minimum size of the total suspended message buffers from $(\log_2 P + V) \times rq_{max} \times P^2$ to $P \log_2 P$ where V is the size of a memory block in scale of bits. Table 6.1 summarizes the hardware cost to avoid deadlock in two systems. Table 6.2 shows the hardware cost to avoid livelock in two systems. Since livelock occurs with extremely low probability because of variety of a latency of message passing, a designer may not spend the cost in Table 6.2 to guarantee that the livelock will never occur. Therefore, this hardware cost is optional. Because P and N are much larger than A, V, M, and rq_{max}, BCachet requires much smaller hardware cost than the original Cachet.

	BCachet	**Cachet**
The Size of Total Message Queues (number of messages)	$P(rq_{max} + 6)$	$NP + 3P$
The Size of A Suspended Message Buffer (bits)	$P \log_2 P$	$(\log_2 P + V) rq_{max} P^2$

P : the number of sites
N : the number of address lines
V : the size of a memory block(bits)
rq_{max} : the maximum number of requests per site

Table 6.1: Comparison of Minimum Hardware Costs to Avoid Deadlock Between Cachet and BCachet

From the system designer's point of view, BCachet has some architectural advantages. First, in BCachet, the soundness and the liveness are almost independent to the sizes of the

suspended message buffer and the stalled message queue as well as the directory, so that a designer can easily make a decision on a trade-off between hardware cost and performance. The BCachet system is sound and live, as long as the system meets the minimum constraint about the sizes of these units, which is pretty small compared to moderate buffer sizes. A designer can parameterize these sizes and find optimal trade-off points by simulation without affecting the soundness and the liveness of the system. For example, since the minimum size of the suspended message buffer is one Byte for a DSM system with 2^8 sites, we can find the optimal suspended message buffer size, by changing the size of it in the range over one Byte and simulating the system.

Second, BCachet is very scalable since the minimum sizes of buffers are almost linear in reasonable P range. Since M is usually much larger than $\log_2 P$, the hardware cost for buffer management is approximately linear to the number of sites. Since the hardware cost C for deadlock avoidance is as shown in Formula (6.1) and M is usually much larger than $\log_2 P$, C is approximately linear as shown in Formula (6.2). For example, if we have a DSM which has 2^8 nodes and a message size is 64 Bytes, then $\log_2 P = 8$ and $M = 512$ and the assumption about the sizes M and $\log_2 P$ holds.

$$C = P\{M(rq_{max} + 7) + A(rq_{max} + 1) + 3\log_2 P\} \quad (6.1)$$

$$\approx P\{M(rq_{max} + 7) + A(rq_{max} + 1)\} \quad (6.2)$$

	BCachet	**Cachet**
The Total Size of Fairness Control Units (bits)	$P(rq_{max}(A+1) + 2\log_2 P + M)$	0

P : the number of sites
A : the size of address (bits)
M : the size of message (bits)
rq_{max} : the maximum number of requests per site

Table 6.2: Comparison of Minimum Hardware Costs to Avoid Livelock Between Cachet and BCachet

6.2 Future Work

Further Optimization

The constraint about the size of the low priority output message queue of a cache site can be reduced to 1 by using message holding techniques. Because we have used 4 bits to represent 14 cache states, we can add two new cache states without increasing the number of bits for encoding cache states. We may add $(WbPending, Wb)$ and $(CachePending, CacheReq)$ to represent $WbPendig$ that holds Wb message and $CachePending$ that holds $CacheReq$ message. In this modification, whenever a cache block wants to send a Wb message and the low priority output message queue is full, a cache can set the cache state as $(WbPending, Wb)$. The Wb message held in the cache state can be released if the queue has space later. This modification effectively merges the low priority output message queue into the cache and reduces the minimum size of LOUT to one.

Heuristic Policies

Cachet, and consequently BCachet has enormous adaptivity for various access patterns. Voluntary rules provide options for a system so that the system intelligently chooses one of the options. However, the concrete policy of this choice has not been suggested yet. One idea about heuristic policies is to use hardware to monitor an access pattern and make a decision based on the monitoring result. Mukherjeee and Hill [10] presented a way to use hardware to predict a message stream and use it in speculative operation. They used a history table as used in branch prediction [12]. BCachet may adopt this idea in its heuristic policy. For example, we can use a history table to predict the next input message and can fire a voluntary rule in advance of the input message.

Bibliography

[1] Xiaowei Shen, Arvind, Larry Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. *Computation Structures Group Memo 413*, October 1998.

[2] A. Agarwal, R. Simmon, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on computer Architecture*, pages 280-289, May 1988.

[3] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-21(12):1112-1118, Dec. 1978.

[4] F. Badder and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[5] Xiaowei Shen, Arvind, and Larry Rudolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems *Computation Structures Group Memo 414*, October 1998.

[6] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on computer Architecture*, June 1995.

[7] Xiaowei Shen. Design and Verification of Adaptive Cache Coherence Protocols. *Ph.D. Thesis, EECS, MIT*, February 2000.

[8] Xiaowei Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols *Computation Structures Group Memo 398*, May 2, 1998.

[9] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Supercomputing*, Nov 1994

[10] Shubhendu S. Mukherjeee and Mark D. Hill. Using Prediction to Accelerate Coherence Protocols. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[11] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based Cache Coherence in Large-scale Multiprocessors. *Computer*, pages 49-58, June 1990.

[12] T-Y Yeh and Yale Patt. Alternative Implementation of Two-Level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124-134 1992.

[13] S. Eggers and R. H. Katz. Evaluation the Performance for Four Snooping Cache Coherency Protocols. In *Proceedings of the 16th International Symposium on Computer Architecture*, May 1989.

[14] Richard Simoni. Cache Coherence Directories for Scalable Multiprocessors *Technical Report: CSL-TR-92-550*, October 1992.

[15] W. D. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989.

[16] J. W. Klop. Term Rewriting System. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science,* volume 2. Oxford University Press, 1992.

[17] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.

[18] D. Lenoski, J. Laudon, K. gharachoroloo, A. Gupta, and J. Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148-159, May 1990.

[19] C. Amza, A. cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive Protocols for Software Distributed Shared Memory. *Proceedings of IEEE, Special Issue on Distributed Shared Memory*, 87(3):467-475, Mar, 1999.

[20] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Application for Shared-Memory. *Computer Architecture News*, 20(1):5 44.

[21] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.

[22] S. J. Eggers and R. H. Katz. A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation. in *Proceedings of the 14th International Symposium on Computer Architecture*, May 1987.

[23] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224 234, Apr. 1991.