RACE Protocol:

Remote Access Cache Coherency Enforcement Protocol

V2.0

TM-3100-2000-032

May 2000

Computer System Department
Computer and Software Technology Laboratory
Electronics and Telecommunications Research Institute

Intentionally Left Blank

Abstract

The RACE Protocol is a cache coherency protocol that uses a full-mapped directory-based cache protocol to provide a consistent memory through multi-level memory hierarchy consisting of a number of processor caches (first and second level caches), physically-distributed shared-memory, and third level caches. This document describes a background, a general overview of the RACE protocol, and an implementation utilizing the Intel Standard High Volume system-boards.

Contents

Table of contents . . . i
List of tables . . . iv
List of figures . . . v
Revision history . . . vii

1 Introduction **1**
1.1 Background . . . 1
1.2 Shared-address architectures . . . 1
1.3 Multiprocessors cache coherency . . . 2
1.4 Structure of the report . . . 3
1.5 Historical perspective and acknowledgment . . . 3

2 The RACE Protocol of PDLSystem **5**
2.1 Introduction . . . 5
2.2 Intra-node and inter-node network operations . . . 6
2.2.1 Terminology . . . 6
2.2.2 Intra-node network operations: processor bus operations . . . 6
2.2.3 Inter-node network operations . . . 7
2.3 Memory hierarchy . . . 9
2.3.1 Terminology . . . 9
2.3.2 Processor cache . . . 9
2.3.3 Remote access cache . . . 9
2.3.4 Directory . . . 10
2.3.5 Inter-relationship between memory hierarchy . . . 11
2.3.6 Transactions mapping between intra-node and inter-node networks . . . 11
2.3.6.1 Initiated by processor transaction . . . 11
2.3.6.2 Initiated by in-bound network transaction . . . 12
2.4 The RACE protocol . . . 17
2.4.1 Terminology . . . 17
2.4.2 Assumptions . . . 17
2.4.3 The RACE protocol overview . . . 18
2.4.4 The RACE protocol through network activities . . . 19
2.4.4.1 Local accesses . . . 19
2.4.4.1.1 Coherent local request . . . 19
2.4.4.1.2 Exclusive local request . . . 19
2.4.4.1.3 Invalidate local request . . . 20
2.4.4.1.4 Writeback local request . . . 21

2.4.4.1.5 Write-through local request 21
2.4.4.1.6 Uncached read/write local request 21
2.4.4.2 Remote accesses 22
2.4.4.2.1 Coherent remote request 22
2.4.4.2.2 Exclusive remote request 22
2.4.4.2.3 Invalidate remote request 23
2.4.4.2.4 Writeback remote request 24
2.4.4.2.5 Write-through remote request 24
2.4.4.2.6 Uncached read/write remote request 25
2.4.4.3 Locking support 27
2.4.4.3.1 Local locked access 27
2.4.4.3.2 Remote locked access 27
2.4.4.4 Replacement of RAC blocks 28
2.4.4.5 Concurrent accesses 29
2.4.4.5.1 Multiple exclusive requests 29
2.4.4.5.2 Multiple invalidation requests 29
2.4.4.5.3 Writeback hitting on pending state 30
2.4.4.5.4 Multiple invalidates case 2 32
2.4.4.5.5 Writeback hitting on pending state case 2 32

3 An implementation of the RACE protocol for Intel SHV 34
3.1 Introduction 34
3.2 Intel SHV 34
3.3 A brief review of Pentium® Pro bus protocol 35
3.3.1 Six phases 35
3.3.2 Deferred transaction 36
3.3.3 How the TPA is involved in the FSB activities 37
3.4 Pentium® Pro processor bus transaction types 37
3.4.1 Classification of Pentium® Pro bus transactions 37
3.4.2 Memory types 37
3.4.3 Transfer types 39
3.5 Pentium® Pro cache protocol 43
3.5.1 Cache line states 43
3.5.2 Coherency-related signals 43
3.6 Two steps approach for snooping 44
3.6.1 The first and second steps 44
3.6.2 Snooping tables 44
3.7 Constraints 48
3.8 The RACE protocol for Intel SHV 49
3.8.1 Local memory operations 49
3.8.1.1 Coherent read cases (BRL) 49
3.8.1.2 Exclusive read cases (BRIL) 50
3.8.1.3 Invalidate cases (BIL) 51
3.8.1.4 Writeback cases (BWL) 52
3.8.1.5 Write-through cases (BWTL and BPWT) 52
3.8.1.6 Uncached read/write cases (BUPR and BUPW) 52
3.8.1.7 Locked read and write cases (BLR and BLW) 53

3.8.2 Remote memory operations 55
3.8.2.1 Coherent read cases (BRL) 55
3.8.2.2 Exclusive read cases (BRIL) 56
3.8.2.3 Invalidate cases (BIL) 58
3.8.2.4 Writeback cases (BWL) 59
3.8.2.5 Write-through cases (BWTL and BPWT) 61
3.8.2.6 Uncached read/write cases (BUPR and BUPW) 61
3.8.2.7 Locked read and write cases (BLR and BLW) 62
References 67
Index 68

List of Tables

2.1 Intra-node network transactions 7
2.2 Inter-node network transactions 8
2.3 Processor cache states 11
2.4 Remote access cache states 12
2.5 Directory states 12
2.6 State transition due to processor's local accesses 13
2.7 State transition due to processor's remote accesses 14
2.8 State transition due to in-bound network transactions at the directory 15
2.9 State transition due to in-bound network transactions at the remote access cache 16

3.1 Third-Party control signals (see [6]) 37
3.2 Pentium® Pro bus transactions (see chapter 5 in [3]) 38
3.3 Memory types and their properties (see chapter 11 of [4]) 39
3.4 Pentium® Pro bus transfer types 39
3.5 Pentium® Pro bus transactions 42
3.6 The 1st step decoding table 45
3.7 Snooping table for local accesses 45
3.8 Snooping table for remote accesses 46

List of Figures

2.1 PDLSystem configuration 5
2.2 Local and remote accesses 18
2.3 Coherent local read to DIR_M 19
2.4 Exclusive local read to DIR_S 20
2.5 Exclusive local read to DIR_M 20
2.6 Coherent remote read to RAC_I and DIR_U or DIR_S 22
2.7 Coherent remote read to RAC_I and DIR_M 23
2.8 Exclusive remote read to RAC_S 23
2.9 Exclusive remote read to RAC_I and DIR_U 24
2.10 Exclusive remote read to RAC_I and DIR_S 24
2.11 Exclusive remote read to RAC_I and DIR_M 25
2.12 Invalidate remote request to RAC_S 25
2.13 Uncached read remote request to RAC_I and DIR_U 26
2.14 Uncached write remote request to RAC_I and DIR_U 26
2.15 Locked read and write to a remote location 27
2.16 Remote access cache replacement 28
2.17 Ghost-sharing after replacement of SHARED block 28
2.18 Concurrent exclusive remote reads 30
2.19 Concurrent invalidates 31
2.20 Concurrent writeback and exclusive read 31
2.21 Concurrent writeback and coherent read 32
2.22 Concurrent invalidates with local access 33
2.23 Concurrent writeback and coherent read from the processor 33

3.1 Intel Standard High Volume 34
3.2 Pentium® Pro processor bus transaction phases 35
3.3 Pentium® Pro processor bus deferred transaction 36
3.4 Two steps approach 44
3.5 BRL hits on DIR_M 49
3.6 BRIL hits on DIR_S 50
3.7 BRIL hits on DIR_M 51
3.8 BIL hits on DIR_S 51
3.9 BWTL/BPWT hits on DIR_S 52
3.10 BWTL/BPWT hits on DIR_M 53
3.11 BLR hits on DIR_S 54
3.12 BLR hits on DIR_M 54
3.13 BRL misses on DIR_U or DIR_S at the home directory 55

3.14 BRL misses on DIR_M at the home-directory 56
3.15 BRIL hits on RAC_S 56
3.16 BRIL misses on DIR_U at the home directory 57
3.17 BRIL misses on DIR_S at the home directory 57
3.18 BRIL misses on DIR_M at the home directory 58
3.19 BIL hits on RAC_S 58
3.20 INVq hits on DIR_{PS} 59
3.21 Writeback BWL 60
3.22 RAC writeback due to replacement 60
3.23 Replacement of a block in RAC_S state 61
3.24 BWTL/BPWT hits on DIR_S 61
3.25 BWTL/BPWT hits on RAC_I 62
3.26 BUPR remote request to RAC_I and DIR_U 62
3.27 BUPW remote request to RAC_I and DIR_U 63
3.28 BLR hits on RAC_S 64
3.29 BLR hits on RAC_M 64
3.30 BLR misses on DIR_U at the home directory 65
3.31 BLR misses on DIR_S at the home directory 65
3.32 BLR misses on DIR_M at the home directory 66

Revision History

- 29 May 2000: Version 2.0 (TM-3100-1999-032)
- 22 Oct. 1999: subtitles are added in the section 2.4.4.5.
- 8 June 1999: added more explanation about Figures 2.19 and 2.22 in the section 2.4.4.5; Figure 3.20 in section 3.8.2.3 is added and its explanation is also added
- 26 May 1999: in order to make the point clear, a footnote is added in the second paragraph of the section 3.8.2.1
- 15 Apr. 1999: table 2.6 – “write through” row is corrected; table 3.8 – rows of hitting on PM with HITM driven by other processor are corrected.
- 26 Jan. 1999: Version 1.0 (TM-3100-1999-012)
- 22 Jan. 1999: figures 2.21, 2.22, and 2.23 are added.
- 25 Nov. 1998: uncached accesses never hit on cached region – tables 2.6, 2.7, 2.9, and 2.9 are revised, and sections 2.4.4.2.6 and 3.8.2.6 are also revised.
- 18 Nov. 1998: Version 0.2 is released.
- 16 Nov. 1998: tables 3.7 and 3.8 are revised.
- 10 Nov. 1998: section 3.8.2.7 is revised and figures in that section are also revised.
- 9 Nov. 1998: figures 3.9, 3.10, 3.24, and 3.25 are added; sections 2.4.4.1.6, 2.4.4.2.6, 3.8.1.6, and 3.8.2.6 are added.
- 6 Nov. 1998: table 2.7 is revised; ‘URD’ and ‘UWR’ are added in table 2.8 and 2.9.
- 28 Oct. 1998: section 3.8.1.3 and 3.8.2.3 are revised.
- 27 Oct. 1998: figures 2.12 and 3.19 are added; many figures are revised.
- 26 Oct. 1998: ‘WRT’ network transaction is removed in section 2.2.3; table 2.7 and 2.8 are revised; section 2.4.4.1.5, 2.4.4.2.5, 3.8.1.5, and 3.8.2.5 are revised.
- 21 July 1998: ‘2.3:retry’ from figure 3.11 at page 54 is deleted.
- 21 July 1998: figures 2.4, 2.7, 2.8, 2.10, 2.11, 3.5, 3.6, 3.7, 3.14, 3.15, 3.17, 3.18, 3.22, 3.28, 3.30, 3.31, and 3.32 are revised.

- 10 July 1998: section 2.4.4.3.2 at page 27; the contents are re-written and figure 2.15 is modified.
- 9 July 1998: sections 3.8.1 and 3.8.2; pending states are corrected.
- 9 July 1998: section 3.3.1 at page 35; two signals (`AP[1:0]` and `RP`) are added to 'request phase'.
- 8 July 1998: section 3.6.2 at page 44; a paragraph at the beginning of section 3.6.2 is added.
- 7 July 1998: Version 0.1 is released.
- April 1998: Version 0.0 Draft is presented.
- March 1998: start to design the protocol.

RACE Protocol:
Remote Access Cache Coherency Enforcement Protocol
V2.0[1]
TM-3100-2000-032

Computer System Department
Computer and Software Technology Laboratory
Electronics and Telecommunications Research Institute
161 Kajong-Dong, Yusong-Gu, Taejon 305-350, Korea

29 May 2000

[1]Subject to change. Do not disclose it since Intel confidential information and ETRI-CSTL confidential information are contained.

Chapter 1

Introduction

1.1 Background

In the last few decades, the high-performance computing community has been turning its attention to multiprocessors systems, that utilize high-performance micro-processors commercially available. Programmers would like to all of the data to be accessible from every processors (or processes), meaning that programmers normally prefer the shared-memory programming model rather than the message-passing model, and thus one of the challenges facing multiprocessor systems is to provide the shared-memory programming model in which processes communicate through shared variables. Compare to this, message-passing programming model enables processes to communicate through explicit message passing. With recent technology advancement, there have been many attempts to support shared-address abstraction on the architecture where the basic architecture has physically-distributed memories.

A typical shared-memory architecture is bus-based symmetric multiprocessors, where all processors can access all memory locations through the bus. From hardware point of view, although it is an inexpensive yet effective way of connecting small number of processors, this central memory approach is costly and inefficient for a large number of processors. As a result, there is a tendency of computer architecture in which processors and memories become more physically distributed. More recently, the design trend in high-performance computer architecture is toward larger and more complex processing nodes that physically share memory and packaging.

To tackle the contradiction between physical-tendency and logical-preference, there have been proposed hybrid mechanisms which combine distribution of memories for fast local accesses and provision of additional mechanisms for accessing all of the memory through a network. This allows the compiler and operating system to provide the user with a shared address space. This physically-distributed but logically-shared memory architecture allows the machine to combine the hardware efficiency of distributed memory with the software convenience of a single address space.

1.2 Shared-address architectures

Shared address space provides the programmer with a single memory abstraction and is well suited for application programs that exhibit fine-grain communication. Shared information can be passed by references because data in the shared address space is accessed through read and write operations. The easier programmability of the shared-memory paradigm is mainly due to its ability to relieve the programmer of the burden of explicit management of data.

On the other hand, the distributed memory multiprocessors, such as the Intel Paragon, provide a more difficult programming model called message passing. Users must write message-passing programs that deal with separate address spaces, synchronizing processors, and communications data using explicit messages. Thus, message passing systems require more explicit intervention from the programmer to co-ordinate the movement of data.

Due to its easy programmability, many parallel computers provide a shared address space.

`Uniform-memory-access` (UMA) architecture machines, such as Encore Multimax, Sequent Symmetry, TICOM-II, and TICOM-III, have centralized shared physical memory. All processors contend to the access shared memory via a network. Thus, most UMA systems employ a private cache to solve memory contention.

`Non-uniform-memory-access` (NUMA) architecture machines distribute memory among processors. Physical memory is distributed among processors so that a portion of the memory (local/near memory) is local to each processor and the near memory associated with each processor forms the shared memory for the system. Processor's near memory can be accessed directly. However, the remaining memories (remote memories) are accessed remotely across an interconnection network. To reduce the latency of remote accesses, caches are used, such as Stanford DASH. This kind of NUMA is called `cache-coherent-NUMA` (CC-NUMA). In CC-NUMA systems, the location of an address's home is determined statically from the address and coherency is achieved automatically by hardware. Recently, many CC-NUMA machines such as SGI Origin 2000, Sequent NUMA-Q, Data General AViiON 20000, and HAL S1, are introduced. Some machines, *e.g.* IBM RP3, provide caches that is kept coherent by software. Machines without coherent cache, called non-cache-coherent-NUMA (NCC-NUMA), include BBN Butterfly and Cray Research T3D. In NCC-NUMA systems, the programmer was compelled to copy data to local addresses to optimize performance because addresses has fixed physical locations. Coherency also had to be managed explicitly within the program.

`Network-abstraction-memory architecture` (NAMA) machines, such as Memnet and Capnet, view the network as memory rather than simple input-output or use shared memory as a network abstraction for high performance communications. The hardware of the memory modules manage the network communication transparently.

`Cache-only-memory architecture` (COMA) machines, such as Kendall Square Research KSR-1 and DDM, maintain distributed memory as a cache. Since the local memory is organized and managed like a hardware cache, both `tag` and `state` information must be maintained.

`Shared-virtual-memory architecture` (SVMA) machines, such as Ivy and Munin, provide virtual address space shared by all the processors and memory coherence is maintained entirely in software. SVMA's local memory (*i.e.* node's main memory) is used as a local cache of the machine's global virtual address space.

The rest of this report looks into NUMA architecture.

1.3 Multiprocessors cache coherency

The cache is a small but fast memory whose purpose is to hold currently active data, thereby reducing the number of access to memory. By keeping the active fraction of the address space in fast storage, the program runs as if the whole address space is fast. A cache can be *write-*

through, that is, writes are passed along directly to the memory. In a *writeback* cache, the copy of a line is only updated in the cache. When the updated cache line must be flushed to make space for a newly referenced line, the contents of the line are written back to memory. Normally, the writeback cache uses *write-allocate* scheme in which the cache allocates cache line on write miss, but the write-through cache usually uses *no-write-allocate* scheme.

On a multiprocessors where each processor has its private cache, a cache coherency protocol is used to provide all processors with a coherent view of memory. When a processor changes the value of a variable, the new value is initially written only in that processor's cache, not in shared memory. As a result, during some period of time, that memory location will have an obsolete value of the variable. Therefore, some mechanism, called cache coherency protocol, has to be employed to ensure that other processors do not access the 'stale' value in memory.

Cache coherency protocols can be classified in two large groups: software-based and hardware based. In this report, software-based solutions are out of scope. *Hardware-based solutions* can be principally divided into two schemes: snoopy and directory protocols. In *snoopy protocol*, each cache is able to watch on the network and to recognize the actions and conditions for a coherence violation, which imposes some reactions in order to preserve coherence. In *directory protocol*, the global status information relevant for coherence maintenance is stored in somewhere called directory. Each memory reference cannot resolved locally should be delegated to the directory having the responsibility for coherence enforcement, that is interrogated and necessary coherence actions take place. Directory protocols can be divided further into three groups: full-map, limited, and chained directory schemes. The last two schemes are not discussed in here, since the protocol discussed in this report utilizes full-map scheme. The basic organization of full-map scheme is that the directory is stored in the main memory and contains entries for each memory block. Each directory entry has state bits and presence bits. The state bits keep caching status and the presence bits point to exact locations of every cached copy of the memory block.

1.4 Structure of the report

This document contains the ***RACE*** (Remote Access Cache coherency Enforcement) protocol which enforces data coherency through memory hierarchy of a NUMA consisting of the processor cache, the local memory, the remote access cache, and the remote memory. In chapter 2, the RACE protocol is explained in general and then in chapter 3, an implementation of the RACE protocol for Intel SHV (Standard High Volume) is presented.

1.5 Historical perspective and acknowledgment

Historical perspective

Dr. Woo-Jong Hahn, who was the chief of the CC-NUMA hardware group consisting of the processing node and network teams, proposed a system architecture in general utilizing the Intel SHV in February 1998. Dr. Hahn was also served as the leader of the processing node team and Mr. Sang-Seok Shin became the leader of the network team. At that time network option was still unknown.

In March 1998, Dr. Ando Ki was appointed the leader of the small task group in the processing node team having objectives to design detailed hardware architecture and cache protocol. Some time later in March, the name of the processing node team changed to the CC-Link team and finally in August, it was fixed as Cache Coherent Agent (CCA) team. Some day in May, the decision was made to use Scalable Coherent Interface (SCI: IEEE Std. 1596-1992) as the inter-node network.

As the result of Dr. Ki's single-handed effort the bulk of the text of the RACE protocol first appeared in April 1998 as the Version 0.0 Draft. It was not a whole set of specification since it contained general protocol based on physically-distributed but logically-shared (PDLS) system. Many presentations and discussions were followed and without these activities this document would not have been possible. Especially Mr. Sang-Man Moh was actively providing invaluable ideas and helped shape this document.

After releasing Version 0.0 Draft, many problems came visible in order to adapt the protocol to combination of the Intel SHV and SCI (more specifically Dolphin LinkController™ chip). The main obstacle was lack of information. However, Mr. Sang-Seok Shin and Mr. Seong-Woon Kim, who had background of Intel Pentium® Pro processor system and Mr. Won-Se Sim and Mr. Jong-Seok Han, who did their best to obtain information of LinkController™ chip made it possible to the RACE protocol Version 0.1 in July 1998.

Many problems still remained until the RACE protocol Version 0.2 was released in November 1998. One particular issue was of locking related and it was solved with the aid of many people who are Mr. Jae-Kyung Lee, Mr. Sung-In Jung, Ms. Sun-Ja Kim, Ms. Jeong-Nyeo Kim, and Mr. Il-Yeon Cho, in CC-NUMA operating system group led by Mr. Hae-Jin Kim.

Acknowledgment

- The following people are acknowledged, who have contributed to the RACE protocol: Woo-Jong Hahn, Ando Ki, Seong-Woon Kim, Sang-Man Moh, and Kyu-Hyun Shim.[1]

- The contributions of others are also acknowledged: Sung-Hun Choi, Jong-Seok Han, Kyoung Park, Sang-Seok Shin, Won-Se Sim, and Chul-Ho Won.[2]

- Any comments can be directed to Ando Ki (`adki@computer.etri.re.kr`) or Sang-Man Moh (`smmoh@computer.etri.re.kr`).

[1] 기 안도, 김 성운, 모 상만, 심 규현, 한 우종

[2] 박 경, 신 상석, 심 원세, 원 철호, 최 성훈, 한 종석

Chapter 2

The RACE Protocol of PDLSystem

2.1 Introduction

The ***PDLSystem*** (Physically-Distributed but Logically-Shared memory system) a kind of CC-NUMA consists of a number of processing nodes that are interconnected via an inter-node network as shown in figure 2.1. The processing node contains processors with private caches, a memory module which is a part of globally-shared memory space, an I/O, and a network interface containing a remote access cache and a directory.

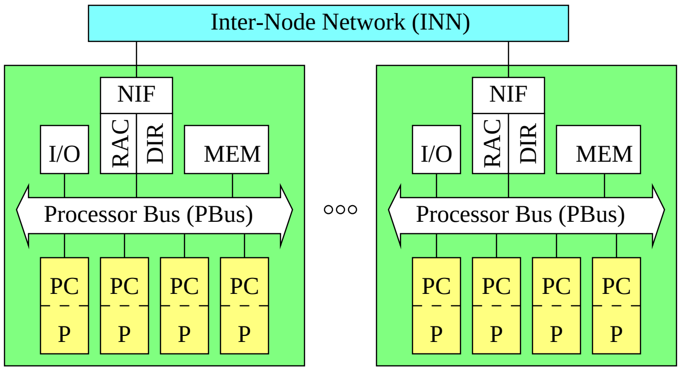


Figure 2.1: PDLSystem configuration: Processor (P); Processor Cache (PC); Network Interface (NIF); Directory (DIR); Remote Access Cache (RAC)

The processor caches within the processing node are kept coherent by an invalidation-based snoopy cache coherency protocol. An access of a location not already in the processor cache generates a request on the processor bus to fetch the corresponding cache line. If the location is mapped on the local (near) memory within the processing node, the memory usually provides the requested data. Otherwise, the request is handled by the remote access cache; *i.e.* the remote access cache provides the data when it contains the data in valid state, otherwise fetching the data from the remote (far) memory where the location is mapped through the inter-node network. Accesses from the outside of the processing node are tracked by the directory to keep caching (sharing) information.

This chapter explains the ***RACE*** (Remote Access Cache coherency Enforcement) protocol in the context of general directory-based ***CC-NUMA*** (Cache-Coherent Non-Uniform Memory Access) architecture.

2.2 Intra-node and inter-node network operations

2.2.1 Terminology

In order to describe the network operations, the following terms are used.

- **Requestor** initiates the first request. The processor is a typical example and it starts a read or write request that is followed by many necessary operations.
- **Agent** performs necessary operations to accomplish the requestor's needs. The remote access cache and the directory can be agent.
- **Transaction** is a series of operations where any operation cannot be missing.
- **Operation** is an atomic so that it cannot be divided. For example, a network operation is just a message.

Other yet undefined terms can be found in section 2.4.

2.2.2 Intra-node network operations: processor bus operations

Intra-node network transactions are as follows.

- **coherent read** (BusCoRd): It retrieves read-only copy of a cache line. The requestor does not intend to modify the line when the line is returned.
- **exclusive read** (BusExRd): It retrieves writable-exclusive copy of a cache line. The requestor intends to modify the line when the line is returned.
- **invalidate** (BusInv): It invalidates all copies at other caches. The requestor contains the line and intends to modify the line.
- **writeback** (BusWrb): It writes a cache line to the memory. The requestor evicts the line due to replacement and no longer contains the line.
- **write through** (BusWrt): It writes a cache line or part of it to the memory. The requestor may or may not contain the data and updates the memory.
- **locked read** (BusLocRd): The requestor intend to perform a read-modify-write operation and it is the first read part.
- **locked write** (BusLocWr): The requestor finishes a read-modify-write operation and it is the last write part.
- **uncached read** (BusUcRd): It is a normal read. The requestor does not intend to cache the data when the data is returned.
- **uncached write** (BusUcWr): It is a normal write. The requestor does not contains the data and updates the memory.

- **input/output read** (BusIoRd): It is a read to I/O-mapped address space. The requestor does not cache it.
- **input/output write** (BusIoWr): It is a write to I/O-mapped address space. The requestor does not cache it.

In the context of cache protocol, intra-node network transactions can be classified as shown in table 2.1.

Table 2.1: Intra-node network transactions

class	transaction name			
	read		write	
cached	coherent read exclusive read invalidate	BusCoRd BusExRd BusInv	writeback write through	BusWrb BusWrt
uncached	uncached read	BusUcRd	uncached write	BusUcWr
locked	locked read	BusLocRd	locked write	BusLocWr
input/output	input/output read	BusIoRd	input/output write	BusIoWr

It is assumed that the intra-node network is a bus, whose data transfer protocol is an idealized split protocol consisting two transfer cycles called request and respond cycles. The request cycle carries address (and data for write transaction) related information and the respond cycle returns reply (and data for read transaction) information. One particular example can be found in [8].

2.2.3 Inter-node network operations

There are three basic classes of inter-node network transactions: cached, uncached, and input/output transaction classes.

The *cached transactions* are used to support reads and writes to cacheable address space and consist of 'coherent read', 'exclusive read', 'invalidate', 'writeback', and 'write through'.

- **coherent read** (CRD): It retrieves a read-only copy.
- **exclusive read** (ERD): It retrieves a writable exclusive copy.
- **invalidate** (INV): It forces cached copies at other agents to invalidate.
- **writeback** (WRB): It updates the home memory and is usually generated as a result of replacement.

The *uncached transactions* are used to support reads and writes by requestors that do not intend to cache the data and consist of 'uncached read' and 'uncached write'. Requestors having no caches use these as well as requestors having caches can use these transactions for uncacheable address space such as memory-mapped I/O.

- **uncached read** (URD): It retrieves the most up-to-date data and the issuing agent may or may not contain the resultant data: *i.e.* when reading from memory, invalid blocks are never filled, but valid blocks return the data without making network transactions.

- **uncached write** (UWR): It updates a cache block when possible and writes through the home memory: *i.e.* when writing through to memory, invalid blocks are never filled, and valid blocks are updated.

It is not recommended that an address space is accessed by both cached transactions and uncached transactions. However, the RACE protocol assumes that bus agents without cache use these uncached transactions to access cacheable address space.

The *input/output transactions* are used to support I/O-mapped reads and writes by requestors that do not intend to cache the data and consist of 'input/output read' and 'input/output write'.

- **input/output read** (IOR): It simply retrieves the specified I/O-mapped memory location.

- **input/output write** (IOW): It simply updates the specified I/O-mapped memory location.

From the coherency protocol perspective, the input/output transactions are never visible. Therefore agents merely forward these transactions to the home node.

By defaults, each network transaction consists a request operation and a reply operation and summarized in table 2.2. Here suffixes 'q' and 'p' are used to indicate request and response.

Table 2.2: Inter-node network transactions

class	transaction name	operation	
		request	reply
cached	coherent read	CRDq	CRDp†
	exclusive read	ERDq	ERDp†
	invalidate	INVq	INVp
	writeback	WRBq⋆	WRBp
uncached	uncached read	URDq	URDp†
	uncached write	UWRq⋆	UWRp
input/output	input/output read	IORq	IORp†
	input/output write	IOWq⋆	IOWp

Remarks: 1) Operations with ⋆ carry data.
2) Operations with † usually carry data, but may not.

In principle, each network transaction begins with a request operation that may initiate other necessary network transactions and ends with a reply operation. For example, an exclusive read request may be followed by invalidate transactions and then finally an exclusive read reply terminates the transaction.

Each reply operation has attribute bits that determine detailed meaning. For example, each reply can turn into a negative acknowledgement to inform error or retry condition. For simplicity, the terms 'RPL' and 'NAK' will be used instead of indicating each different reply when interpretation is obvious, where the former indicates the default reply and the latter negative acknowledgement. The negative acknowledgement reply must contain sufficient information about the reason why its request has been negative acknowledged, such as hitting on pending state, buffer full, or transmission error.

2.3 Memory hierarchy

2.3.1 Terminology

- **valid**: This line is valid at its memory hierarchy.
- **clean**: The memory copy at its home node is valid.
- **exclusive**: This line is only valid at its memory hierarchy.
- **line**: It is a data unit in the processor cache.
- **block**: It is a data unit in the remote access cache and directory. It should be noted that the size of the block is equal to or larger than that of the line.

2.3.2 Processor cache

A kind of MESI protocol [9] is assumed in the processing node to keep the processor caches coherent. The processor cache is writeback cache.

- **I** (INVALID, PC_I): The line is not available in the cache. A processor access to this line misses the cache and can cause the processor cache to fetch the line into the cache from the processor bus (from memory or from another cache including other processor cache or the remote access cache).
- **S** (SHARED, PC_S): The line is in the cache, contains the same value as in the memory, and can be SHARED state in other caches. Processor-reading the line causes no processor bus activity. Processor-writing the line causes an invalidate processor bus operation to gain writable exclusive ownership of the line.
- **E** (EXCLUSIVE, PC_E): The line is in the cache, contains the same value as in the memory, and is INVALID in all other caches. Processor-reading the line causes no processor bus activity. Processor-writing the line causes no processor bus activity, but changes the line's state to MODIFIED.
- **M** (MODIFIED, PC_M): The line is in the cache, contains a more recent value than the memory, and is INVALID in all other caches. Processor-reading or processor-writing the line causes no processor bus activity.

2.3.3 Remote access cache

From the processor's point of view, the remote access cache behaves like a memory providing remote data to the processor, but it actually works as a cache. As far as remote addresses are concerned, *space inclusion*[1] is preserved between the remote access cache and the processor caches in the same processing node. Each block can be one of the following states.

[1]The *space multi-level inclusion* property [1] assumes that the cache lines in the higher level (*e.g.* processor cache) are a subset of that of the lower level cache (*e.g.* remote access cache), but might not be equal in value.

- **I** (INVALID, RAC_I): The block is not available in the cache. A processor bus access to this block misses and can cause the remote access cache to fetch the block into the cache from the network (from remote memory).
- **S** (SHARED, RAC_S): The block is a read-only copy and the same as its home memory. None, one, or more remote access caches may have a read-only copy.
- **M** (MODIFIED, RAC_M): The block is a writable exclusive copy and contains a more recent value than its home memory.
- **L** (LOCKED, RAC_L): The block contains a value interlocked, and is INVALID in all other remote access caches. Any processor bus accesses except locked read and locked write are not allowed to access this block, but the locked write changes its state to MODIFIED (RAC_M).

In addition to these states, there is a pending bit that indicates state in transition, meaning that the remote access cache tag is not updated atomically – a cache state is set to an intermediate value when a packet is sent out and is set to its final value when the response to that packet is received. For example, PM indicates the block is in the middle of transition from M. The block in pending state at the remote access cache indicates that a previous request coming to the remote access cache for that block is still in progress and has not been completed.

When a block needs to be written back upon a miss (explicit write back by replacement), the block is written back first and then the miss is satisfied. The remote access cache only writes back the RAC_M block, so *replacement-hint* for replacement of RAC_S block is not used and as a result the *ghost-sharing* state in the directory is allowed.

When the remote access cache provides a clean sharable data to the processor cache, it must be ensured that the processor cache does not have the data in exclusive state, since the processor cache can modify the line in PC_E and changes it to PC_M without notification (*i.e.* invalidation) meaning that state incompatibility occurs between the processor cache and the remote access cache.

The remote access cache acts as both a pseudo-memory controller that translates access to remote data as well as a pseudo-processor that retrieves remote data in the processor cache.

2.3.4 Directory

Each entry of directory consists of a number of presence bits and state bits. The presence bits point to sharing nodes. Each entry of directory can be one of the following states in order to indicate the sharing states among remote access caches.

- **U** (UNCACHED, DIR_I): The memory block is valid and no other remote access caches have the copy.
- **S** (SHARED, DIR_S): The memory block is valid and none, one, or more remote access caches may have read-only copies.
- **M** (MODIFIED, DIR_M): The memory block is not available in the home memory and a remote access cache has a writable exclusive copy.

In addition to these states, there is a pending bit that indicates state in transition, meaning that the directory state is not updated atomically – a state is set to an intermediate value when a packet is sent out and is set to its final value when the response to that packet is received. For example, PM indicates this block is in the middle of transition from M. The pending bit enables the directory controller to change the proper state transition. A block being in pending state at the directory indicates that a previous request coming to the home for that block is still in progress and has not been completed.

The directory does not keep track of local processor sharing: *i.e.* when a processor in the local node caches a data, the presence bit would not indicate it.

2.3.5 Inter-relationship between memory hierarchy

Tables 2.3, 2.4, and 2.5 show inter-relationship among other memory hierarchy. It should

Table 2.3: Processor cache states

for remote addresses

state	valid	clean	exclusive	remote access cache at local node		directory at remote home node	
				possible	impossible	possible	impossible
I	no	–	–	I,S,M,L	–	U,S,M	–
S	yes	yes	no	S	I,M,L	S	U,M
E	yes	yes	yes	M	I,S,L	M	U,S
M	yes	no	yes	M	I,S,L	M	U,S

for local addresses

state	valid	clean	exclusive	directory at local home node		remote access cache at remote node	
				possible	impossible	possible	impossible
I	no	–	–	U,S,M	–	I,S,M,L	–
S	yes	yes	no	U,S	M	I,S	M,L
E	yes	yes	yes	U	S,M	I	S,M,L
M	yes	no	yes	U	S,M	I	S,M,L

be noted that the directory must notify the processor cache of sharing state in order for the processor cache not to get a copy in PC_E. See section 2.3.3 for details.

2.3.6 Transactions mapping between intra-node and inter-node networks

2.3.6.1 Initiated by processor transaction

Tables 2.6 and 2.7 shows that how processor bus transactions incur which network transactions and change states of memory hierarchy.

Inter-node input/output transactions are not considered in this section since it is assumed that the RACE Protocol does not utilize them. However, it is not difficult to map inter- and intra-node input/output transactions because they can be treated another kind of uncached transactions.

Table 2.4: Remote access cache states

state	valid	clean	exclusive	processor cache states at local node		directory states at home node	
				possible	impossible	possible	impossible
I	no	–	–	I	S,E,M	U,S,M	–
S	yes	yes	no	I,S	E,M	S	U,M
M	yes	no	yes	I,E,M	S	M	U,S
L	yes*	yes	yes*	I	S,E,M	M	U,S

* valid only for locked operations.

Table 2.5: Directory states

state	valid where		processor cache at local home node		remote access cache at remote node	
	local	remote	possible	impossible	possible	impossible
U	yes	no	I,S,E,M	–	I	S,M,L
S	yes	yes/no*	I,S	E,M	I,S	M,L
M	no	yes	I	S,E,M	I,M,L	S

* due to ghost-sharing state

Table 2.6 shows mapping due to the processor's local accesses; as the directory detects a local transaction on the bus, the directory generates necessary network transaction according to its state. For example, a local coherent read hitting on the directory marked as uncached (U) or shared (S) does not generate any network transaction. A local coherent read of modified (M) incurs a coherent read (CRD) network transaction, which goes to the remote access cache containing the line in modified (M) state, and, finally, the directory and the remote access cache change their states to shared (S) and shared (S), respectively.

Table 2.7 shows mapping due to the processor's remote accesses; as the remote access cache detects a remote transaction on the bus, the remote access cache generates necessary network transaction according to its state. For example, a remote coherent read hitting on the remote access cache marked as shared (S) or modified (M) does not generate any network transaction. A remote coherent read of invalid (I) incurs a coherent read (CRD) network transaction, which goes to the home containing the line in one of uncached (U), shared(S), modified (M) states, and, finally, the remote access cache and the directory change their states to shared (S) and shared (S), respectively.

2.3.6.2 Initiated by in-bound network transaction

Tables 2.8 and 2.9 shows state transition due to in-bound network transactions.

Table 2.6: State transition due to processor's local accesses

processor's bus access	current state DIR	possible network action by DIR	possible current state RAC[3)]	next (final) state DIR	next (final) state RAC	remarks
coherent	U	none	I	U	I	
read	S	none	S	S	S	
(BusCoRd)	M	CRD	M	S	S	1)
exclusive	U	none	I	U	I	
read	S	INV	S	U	I	
(BusExRd)	M	ERD	M	U	I	1)
invalidate	U	none	I	U	I	
(BusInv)	S	INV	S	U	I	
	M	ERD	M	U	I	may not occur; 1)
writeback	U	none	I	U	I	
(BusWrb)	S	INV	S	U	I	may not occur
	M	ERD	M	U	I	may not occur; 1); 2)
write	U	none	I	U	I	
through	S	INV	S	U	I	
(BusWrt)	M	ERD	M	U	I	may not occur; 1); 2)
locked	U	none	I	U	I	
read	S	INV	S	U	I	
(BusLocRd)	M	ERD	M	U	I	
locked	U	none	I	U	I	
write	S	–	S	U	I	must not occur
(BusLocWr)	M	–	M	U	I	must not occur
uncached	U	none	I	U	I	
read	S	–	S	S	S	must not occur
(BusUcRd)	M	–	M	M	M	must not occur;
uncached	U	none	I	U	I	
write	S	–	S	U	S	must not occur
(BusUcWr)	M	–	M	U	M	must not occur;

1) need writeback
2) need merging
3) 'L' is not shown since all network transactions to 'L' must retry.

Table 2.7: State transition due to processor's remote accesses

processor's bus access	current state	possible network action by RAC	possible current state	next (final) state		remarks
	RAC		DIR	RAC	DIR	
coherent read (BusCoRd)	I	CRD	U/S/M	S	S	
	S	none	S	S	S	
	M	none	M	M	M	
	L	–	M	L	M	must retry
exclusive read (BusExRd)	I	ERD	U/S/M	M	M	
	S	INV	S	M	M	
	M	none	M	M	M	
	L	–	M	L	M	must retry
invalidate (BusInv)	I	ERD	U/S/M	M	M	may not occur
	S	INV	S	M	M	
	M	none	M	M	M	
	L	–	M	L	M	must not occur
writeback (BusWrb)	I	ERD	U/S/M	M	M	may not occur
	S	INV	S	M	M	may not occur
	M	none	M	M	M	
	L	–	M	L	M	must not occur
write through (write) (BusWrt)	I	ERD	U/S/M	M	M	
	S	INV	S	M	M	
	M	none	M	M	M	
	L	–	M	L	M	must retry
locked read (BusLocRd)	I	ERD	U/S/M	L	M	
	S	INV	S	L	M	
	M	none	M	L	M	
	L	–	M	L	M	
locked write (BusLocWr)	I	–				must not occur
	S	–				must not occur
	M	–				must not occur
	L	none	M	M	M	
uncached read (BusUcRd)	I	URD	U/S/M	I	U	
	S	–	S	S	S	must not occur
	M	–	M	M	M	must not occur
	L	–	M	L	M	must retry
uncached write (BusUcWr)	I	UWR	U/S/M	I	U	
	S	–	S	S	U	must not occur
	M	–	M	M	M	must not occur
	L	–	M	L	M	must not occur

Table 2.8: State transition due to in-bound network transactions at the directory

in-bound network transaction from RAC1	current			bus action by DIR	network action by DIR	next state			remarks
	state of DIR at home	possible state				DIR	RAC1	RAC2	
		RAC1	RAC2[1)]						
CRD	U	I	I	BusCoRd	none	S	S	I	
	S	I	S	BusCoRd	none	S	S	S	
	M	I	M	BusWrb	CRD	S	S	S	
ERD	U	I	I	BusExRd	none	M	M	I	
	S	I	S	BusExRd	INV	M	M	I	
	M	I	M	none	ERD	M	M	I	
INV	U	I	I	–					must not occur
	S	S	S/I	BusInv	INV	M	M	I	ghost-sharing
	M	I	M	–					must not occur
WRB	U	I	I	–					must not occur
	S	I/S	S	–					must not occur
	M	I	M	–					must not occur
	M	M	I	BusWrb	none	U	I	I	
URD	U	I	I	BusUcRd	none	U	I	I	
	S	I	S	–					must not occur
	S	S	S	–					must not occur
	M	I	M	–					must not occur
	M	M	I	–					must not occur
UWR	U	I	I	BusUcWr	none	U	I	I	
	S	I/S	S	–					must not occur
	M	I	M	–					must not occur
	M	M	I	–					must not occur

Remarks: DIR is recipient, RAC1 stands for requestor and RAC2 for others.
1) 'L' is not shown since all network transactions to 'L' must retry.

Table 2.9: State transition due to in-bound network transactions at the remote access cache

in-bound network transaction from DIR	current			bus action by RAC1	network action by RAC1	next state			remarks
	state of RAC1 1)	possible state				RAC1	DIR	RAC2	
		DIR	RAC2						
CRD	I			–					must not occur
	S	S	I/S	–					must not occur
	M	M	I	BusCoRd	none	S	S	S	
ERD	I			–					must not occur
	S	S	I/S	–					must not occur
	M	M	I	BusExRd	none	I	M	M	
INV	I	S	S	none	none	I	M	M	ghost-sharing
	S	S	I/S	BusInv	none	I	M	M	
	M	M	I	–					must not occur
URD	I								must not occur
	S								must not occur
	M		I						must not occur
UWR	I								must not occur
	S								must not occur
	M		I						must not occur

Remarks: RAC1 stands for recipient and DIR for requestor.
1) 'L' is not shown since all network transactions to 'L' must retry.

2.4 The RACE protocol

This section provides a brief high-level introduction to the RACE protocols and thereafter a detailed explanation.

2.4.1 Terminology

In order to describe the protocol, the following terms are used.

- *Local node* or *requesting node* is the processing node containing the processor that issued the memory request.
- *Home node* is the processing node that contains the physical memory associated with a given memory address. *Remote node* is any other processing node.
- *Local memory* is the memory whose home is in the local node. *Remote memory* is the memory whose home is in the remote node.
- *Owning node* is the processing node that holds the block in modified state. Otherwise, the home node is the owning node.
- *Local request* (or *local access*) is the access whose address is mapped on the local memory. *Remote request* (or *remote access*) is the access whose address is mapped on the remote memory.

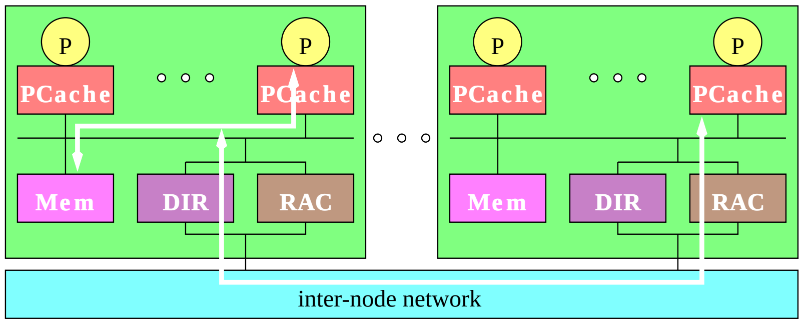
2.4.2 Assumptions

The coherency protocol assumes the following conditions.

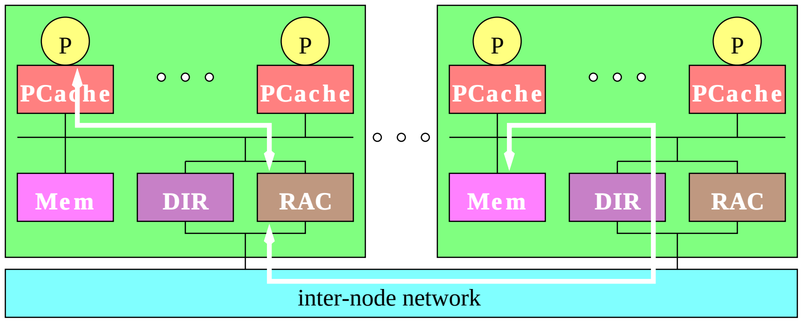
- All remote accesses missing on the remote access cache first are forwarded to the home node.
- The home node serializes accesses to determine their global order.
- The reply-forwarding is used; *i.e.* a request to modified block is forwarded to the owning node by the home node. The owning node replies directly to the local node to provide the requested block and to the home node to update the directory and home memory correctly, as well. However, negative acknowledgement should be reported to the home node only.
- The data transfer protocol through the inter-node network must guarantee that the ordering of messages must be kept such that all messages sent from the source must arrive at the destination in the order they sent, if source and destination are the same: *i.e.* the network preserves point-to-point order of messages between end-points. In other words, the protocol described here relies on in-order message delivery between any two processing nodes, which insures that multiple messages sent by a processing node do not arrive at the same destination node in different order.
- The ghost-sharing is allowed; *i.e.* the remote access cache can replace blocks in shared state without notifying the home node of it. Therefore any invalidation requests to nonresident blocks in the remote access cache must be acknowledged, since the home node still has sharing link (bit map) to non-existing block due to un-notified replacement of shared blocks.

2.4.3 The RACE protocol overview

When a processor incurs a cache miss, the request looks up the address of the memory block that contains the requested cache line, to determine if the home is local or remote. If it is local, the local memory would provide the data to the processor cache (see figure 2.2(a)). Otherwise, the remote access cache would provide the data to the processor cache (see figure 2.2(b)).



(a) Local access



(b) Remote access

Figure 2.2: Local and remote accesses: processor (P); processor cache (PCache); memory (Mem); directory (DIR); remote access cache (RAC)

Although the requested cache line is allocated at the local memory (*i.e.* local request), the directory should be looked up to enforce data coherency because the remote access caches at the remote nodes may have a copy. In order for the remote access cache to provide the requested data (*i.e.* remote request), the remote access cache should have the data in proper state. If not, the remote access cache at the requesting node generates an inter-node request to the home node. There, the directory retrieves the block from its local memory if the home node owns the data. Otherwise the directory at the home node forwards the request to owning node and the owning node sends the data to the requesting node and informs the home node of its completion. Thereafter, the remote access cache at the requesting node finally provides the data to the processor cache.

2.4.4 The RACE protocol through network activities

This section gives an explanation of the RACE protocol from network's point of view. So, a typical MESI protocol for the processor caches is assumed and detailed operations on the processor bus are not given.

The big (shaded) rectangles are the coherent agent, such as the processor cache, the directory, and the remote access cache. The circles (or ovals) in the big rectangles are states. The arcs with boxed labels are network operations and the numbers in the boxed labels show the serialization of operations. Different letters (numbers) next to the same number indicate that the operations can be overlapped.

2.4.4.1 Local accesses

2.4.4.1.1 Coherent local request A coherent local request on the processor bus hitting on DIR_U does not affect the directory and the local memory handles the request. Likewise, a coherent local request on the processor bus hitting on DIR_S does not affect the directory and the local memory handles the request, but the directory must make sure that the processor cache changes its state to PC_S instead PC_E.

A coherent local request on the processor bus hitting on DIR_M makes the directory controller send a coherent read request (CRD) to owning node. On receiving the request, the owning node retrieves most up-to-date data from processor cache within its processing node and then sends it back to the home node. When the directory provides the processor with the data, the memory should be updated to maintain coherent data. It must be ensured that the final state of the processor cache is PC_S. Finally, the directory and the owning node's remote access cache states go to DIR_S and RAC_S as shown in figure 2.3, respectively. It

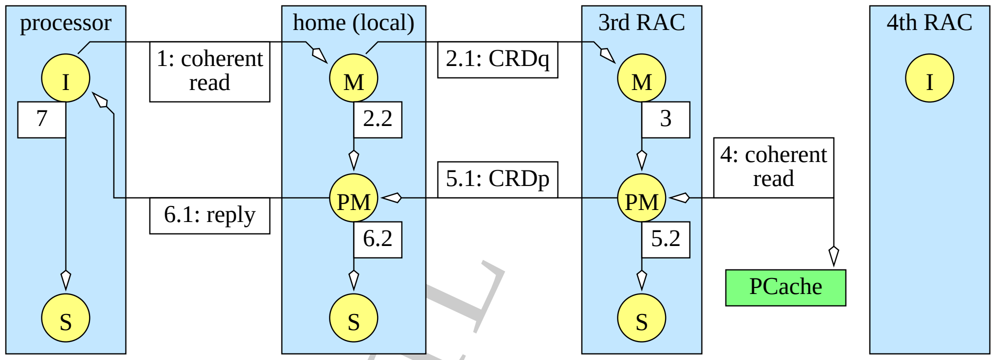


Figure 2.3: Coherent local read to DIR_M

must be noted that if the processor cache line is smaller than the remote access cache block, on receiving coherent read reply, the directory must write the remaining to memory after providing the critical line first to the processor cache.

2.4.4.1.2 Exclusive local request An exclusive local request on the processor bus hitting on DIR_U does not affect the directory and the local memory handles the request.

An exclusive local request hitting on DIR_S makes the directory controller send invalidate requests (INVq's) to all sharing nodes. On receiving the invalidate request, each sharing node invalidates its copy and sends an acknowledgement to the home node. Thereafter each sharing node changes its state to RAC_I. When all invalidate acknowledgements (INVp's) arrive, the directory state changes to DIR_U as shown in figure 2.4. It has to be noted that the directory at the requestor has an exclusive writable ownership just after generating all invalidate requests.

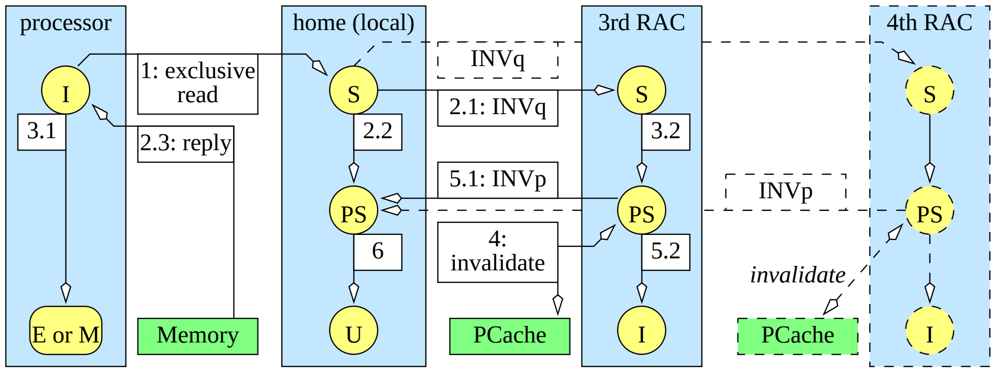


Figure 2.4: Exclusive local read to DIR_S

An exclusive local request hitting on DIR_M makes the directory controller send an exclusive read requests (ERD's) to owning node. On receiving the invalidate request, the owning node evicts (retrieves and invalidates) most up-to-date data from processor cache within its processing node and then sends it back to the home node. Finally, the directory and the owning node's remote access cache states go to DIR_U and RAC_I as shown in figure 2.5, respectively. It must be noted that if the processor cache line is smaller than the remote

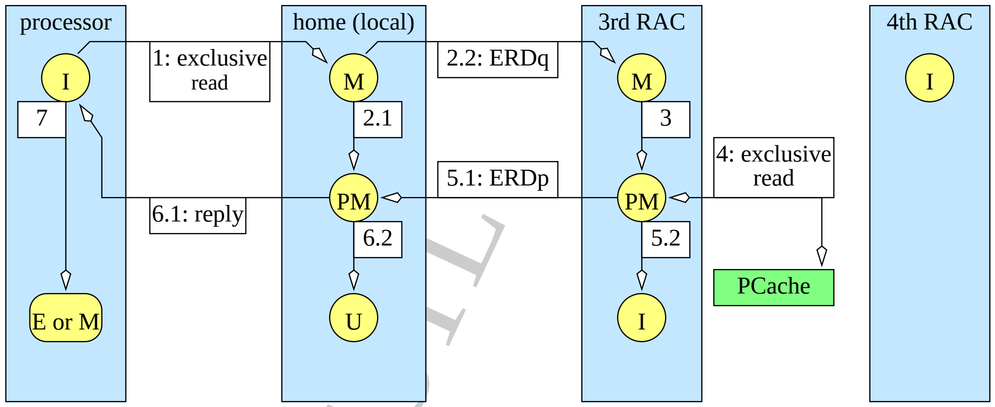


Figure 2.5: Exclusive local read to DIR_M

access cache block, on receiving exclusive read reply, the directory must write the remaining lines after providing the critical line first to the processor cache.

2.4.4.1.3 Invalidate local request An invalidate local request to DIR_U block does not affect the directory and the processor bus operation resolves the request. An invalidate local request to DIR_M block does not occur since DIR_M means that a remote node has the block exclusively.

An invalidate local request to DIR_S is resolved as the same as an exclusive local request to DIR_S. The final reply may not carry data usually, but it can carry data as explained in section 2.4.4.5.

2.4.4.1.4 Writeback local request A writeback local request to DIR_S or DIR_M block does not occur, since the processor generating the writeback must have modified cache line (PC_M) meaning that there is no sharing. As a result, the directory needs not to care of writeback local request.

2.4.4.1.5 Write-through local request A write-through local request to DIR_U and DIR_S is handled as the same as an invalidation local request.

A write-through local request to DIR_M is handled as the same as an exclusive read and, in addition, the memory must merge the data written through and the data gotten by the exclusive read.

2.4.4.1.6 Uncached read/write local request Uncached read/write local request to DIR_S and DIR_M block does not occur, since DIR_S and DIR_M mean that some caching agents accessed the block as cacheable region. As a result, the directory needs not to care of uncached read/write local request.

2.4.4.2 Remote accesses

2.4.4.2.1 Coherent remote request A coherent remote request hitting on RAC_S does not affect its remote access cache and the remote access cache handles the request. Likewise, a coherent remote request hitting on RAC_M does not affect its remote access cache. However, if no processor cache provides the requested data[2], the remote access cache should respond like a memory.

A coherent remote request missing on its remote access cache generates a coherent read request (CRD) to home node. If the state at the home node's directory is DIR_U or DIR_S, the directory retrieves a copy from the memory and sends it back to the requestor as shown in figure 2.6. If the state is DIR_M, the directory forwards the request to owning node. On receiving the coherent request, the remote access cache at the owning node retrieves most up-to-date data from its processor cache and sends it back to both the requestor and the home as shown in figure 2.7. It has to be noted that, on receiving the coherent read reply, the home node performs writeback to update the memory. It must be ensured that the final state of the processor cache is PC_S. When the owning node detects the case of retry or error, the forwarded reply from the owning node to the original requestor must not be sent. Instead the owning node simply notifies the home node of the status. Finally, the remote access cache, the directory, and the owning node's remote access cache states go to RAC_S, DIR_S, and RAC_S, respectively.

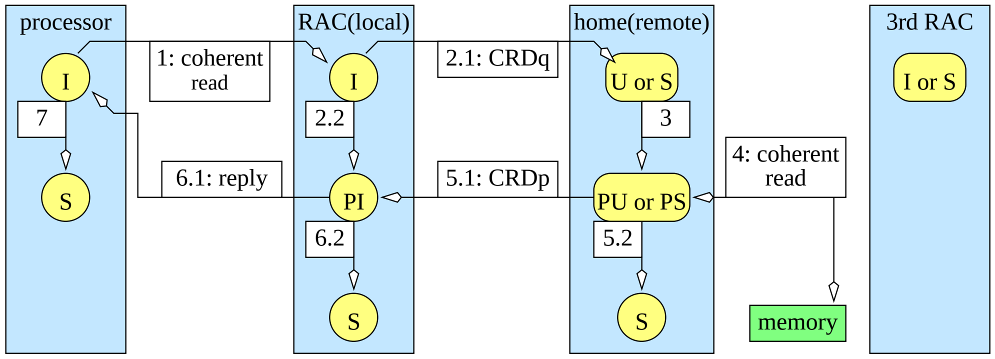


Figure 2.6: Coherent remote read to RAC_I and DIR_U or DIR_S

2.4.4.2.2 Exclusive remote request A exclusive remote request hitting on RAC_M does not affect its remote access cache and the remote access cache handles the request. However, if processor cache provides the requested data, the remote access cache should not reply.

An exclusive remote request hitting on RAC_S makes remote access cache send an invalidate request (INVq) to home node. On receiving the request, the home sends invalidate requests to all sharing nodes and invalidates its processor caches. After sending invalidate requests to all sharing nodes, the directory lets the requestor get an exclusive writable ownership, but it must wait for all invalidate acknowledgements as shown in figure 2.8.

[2]It is assumed that when a processor cache containing modified data sees a request to that data, the processor writes it back. At the same time, the original requesting cache snarfs the data written back to memory.

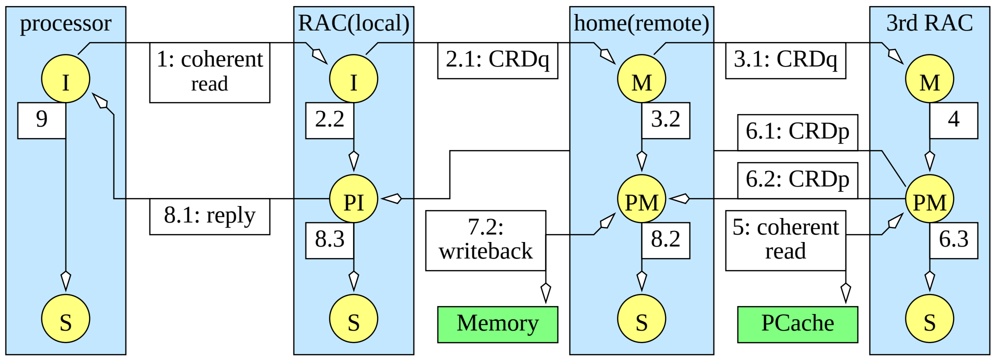


Figure 2.7: Coherent remote read to RAC_I and DIR_M

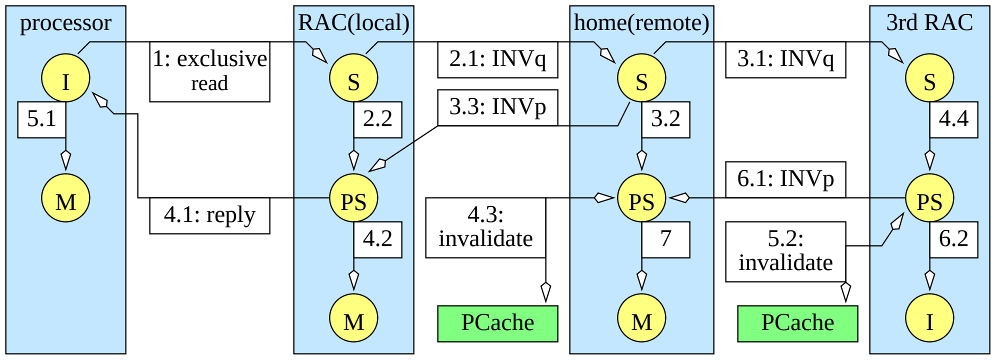


Figure 2.8: Exclusive remote read to RAC_S

An exclusive remote request missing on its remote access cache generates a exclusive read request (ERD) to home node. If the state at the home node's directory is DIR_U, the directory retrieves an exclusive copy from its processor cache and sends it back to the requestor as shown in figure 2.9. If the state is DIR_S, the directory sends invalidate requests to all sharing nodes. At the same time the home retrieves an exclusive copy and sends it back to the requestor as shown in figure 2.10. If the state is DIR_M, the directory forwards the request to owning node. On receiving the exclusive request, the remote access cache at the owning node evicts most up-to-date data from its processor cache and sends it back to the requestor. At the same time the owning node sends a notification of completion to the home as shown in figure 2.11. It has to be noted that the notification reply (`ERDp*` in figure 2.11) does not carry data. When the owning node detects the case of retry or error, the forwarded reply from the owning node to the original requestor must not be sent. Instead the owning node simply notifies the home node of the status. Finally, the remote access cache, the directory, and the owning node's remote access cache states go to RAC_M, DIR_M, and RAC_I, respectively.

2.4.4.2.3 Invalidate remote request An invalidate remote request on RAC_M state at remote access cache does not affect remote access cache state since RAC_M implies the remote access cache already has an exclusive copy. An invalidate remote request on RAC_I state must not occur since RAC_I means processor caches in the processing node do not have copy.

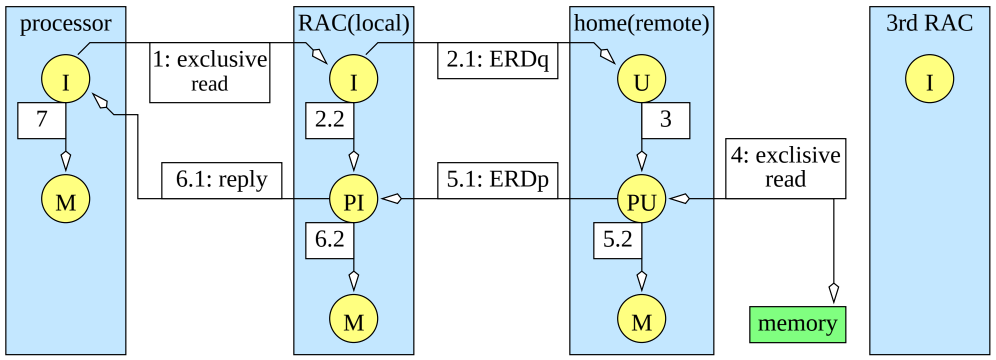


Figure 2.9: Exclusive remote read to RAC_I and DIR_U

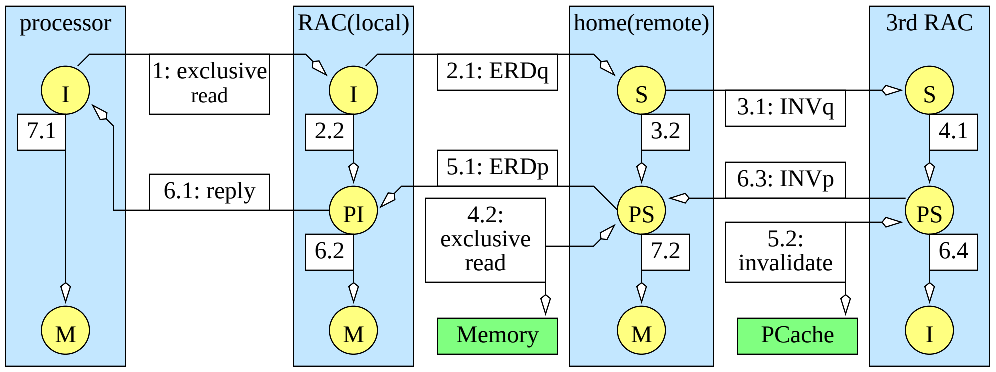


Figure 2.10: Exclusive remote read to RAC_I and DIR_S

An invalidate remote request hitting on RAC_S makes remote access cache send an invalidate request (INVq) to home node. On receiving the request, the home sends invalidate requests to all sharing nodes and invalidates its processor caches. After sending invalidate requests to all sharing nodes, the directory lets the requestor get an exclusive writable ownership, but it must wait for all invalidate acknowledgements as shown in figure 2.12.

2.4.4.2.4 Writeback remote request A writeback remote request to RAC_I or RAC_S does not occur, since the processor cache is not allowed to have PC_M. A writeback remote request to RAC_M just updates its remote access cache and does not propagate it to the home node. More detailed discussion is given in section 2.4.4.4.

2.4.4.2.5 Write-through remote request A write-through remote request on the bus makes the remote access cache work as the same as an exclusive remote request. A write-through remote request on RAC_M updates the remote cache block.

A write-through remote request on RAC_S makes the remote access cache get an exclusiveness and then updates the remote cache block. To do this the remote access cache sends a invalidate request (`INV`) to the home node. On receiving the invalidate request, the home invalidates all sharing nodes and finally the remote access cache and the directory changes their states to modified (M) and modified (M), respectively.

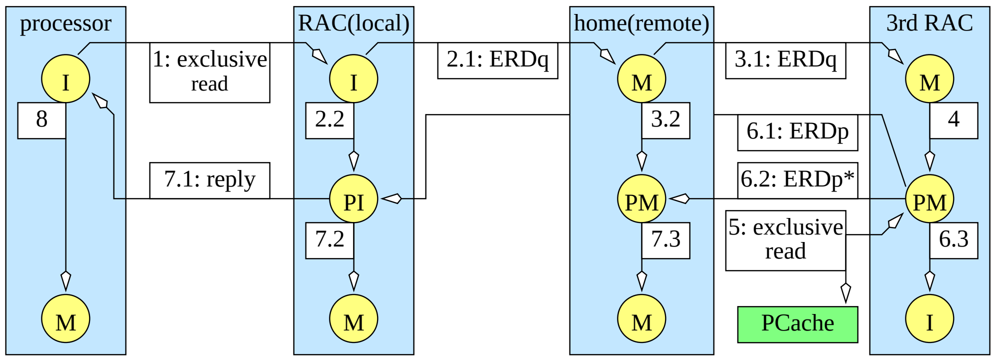


Figure 2.11: Exclusive remote read to RAC_I and DIR_M: `6.2:ERDp*` does not carry data.

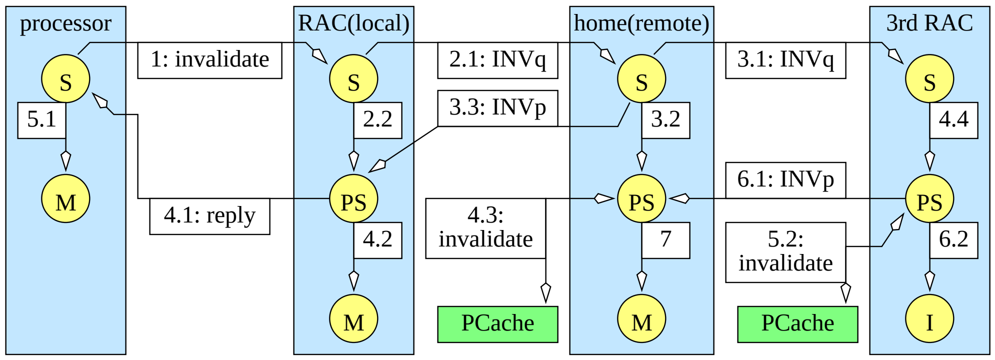


Figure 2.12: Invalidate remote request to RAC_S

A write-through remote request on RAC_I makes the remote access cache get an exclusive copy of the block and then updates it as the same as exclusive remote request does.

2.4.4.2.6 Uncached read/write remote request Uncached read/write remote request to RAC_S and RAC_M block does not occur, since RAC_S and RAC_M mean that the block has been accessed as a cacheable region.

An uncached read remote request generates a uncached read request (URD) to home node. If the state at the home node's directory is not DIR_U, a protocol violation occurs, since DIR_S or DIR_M mean that the block has been accessed as a cacheable region. The directory retrieves a copy from its memory and sends it back to the requestor as shown in figure 2.13.

An uncached write remote request generates a uncached read request (UWR) to home node. If the state at the home node's directory is not DIR_U, a protocol violation occurs, since DIR_S or DIR_M mean that the block has been accessed as a cacheable region. The directory updates its memory and sends the response back to the requestor as shown in figure 2.14.

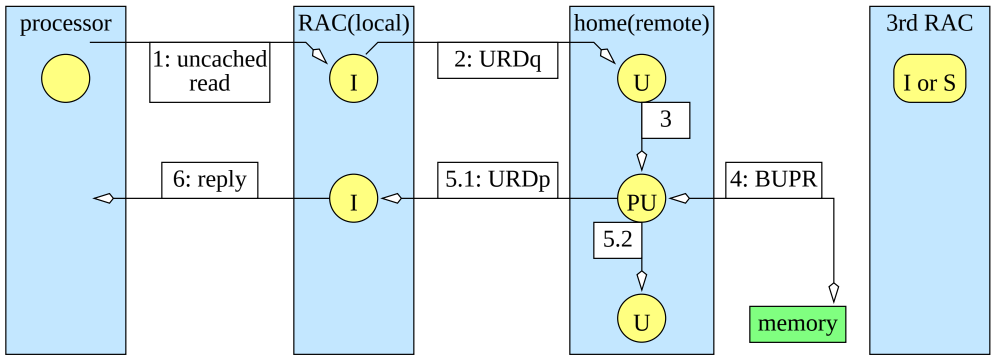


Figure 2.13: Uncached read remote request to RAC_I and DIR_U

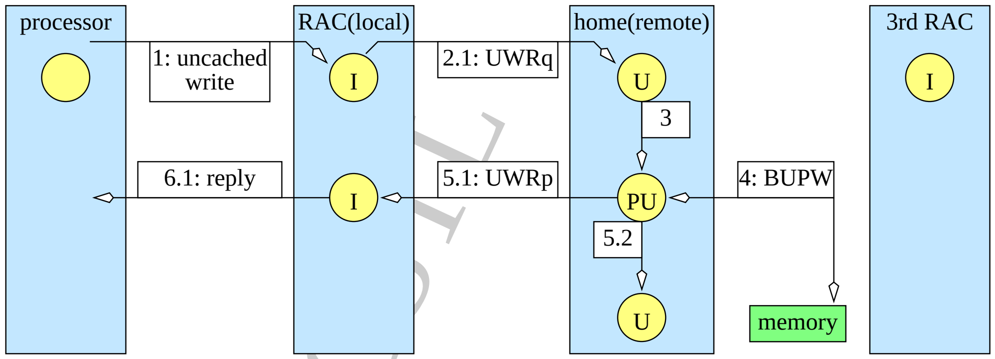


Figure 2.14: Uncached write remote request to RAC_I and DIR_U

2.4.4.3 Locking support

The RACE protocol supports low level synchronizations by utilizing bus locking mechanism in which the processor performs an atomic read-modify-write (RMW) operation to inhibit simultaneous attempts by other processors to alter it. Each atomic RMW operation consists of locked-read and locked-write. It must be guarantee that no other processors can access the specified memory location between these corresponding accesses.

2.4.4.3.1 Local locked access A locked read local request on the bus makes the directory work as the same as an exclusive local request. This ensures that the home node gets an exclusive copy of the requesting block and the bus locking mechanism excludes any further accesses to that memory location except the corresponding locked write. Following locked write may hit on DIR_U or some other pending states, but the bus locking mechanism guarantees un-interruptability between locked read and locked write, so the local memory is updated without any inter-node network operations

2.4.4.3.2 Remote locked access A locked read remote request hitting on RAC_S or RAC_M makes the remote access cache work as the same as an exclusive remote request, except that the final state of the block is RAC_L instead of RAC_M. This ensures that the requesting node gets an exclusive ownership until the corresponding locked write is performed. Following locked write always hits on RAC_L, so the remote access cache is updated without any inter-node network operations as shown in figure 2.15.

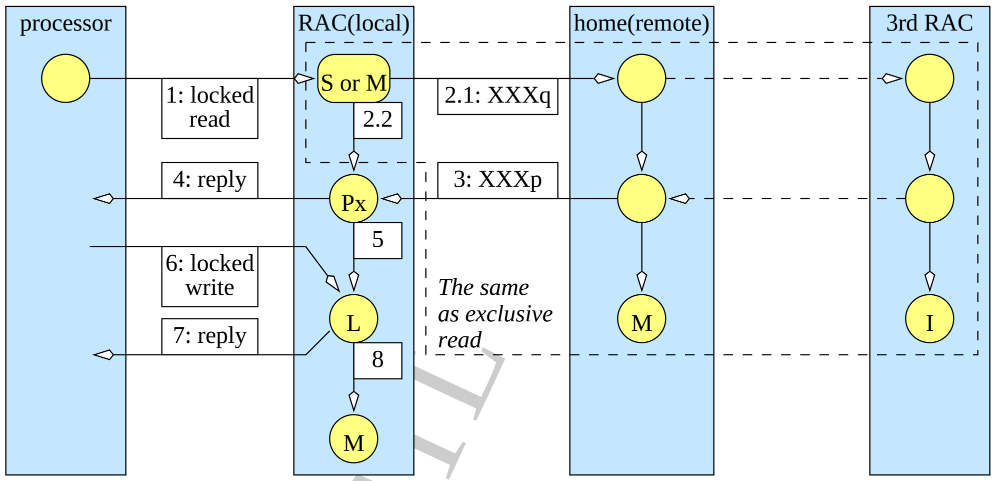


Figure 2.15: Locked read and write to a remote location

A locked read remote request missing on the remote access cache must be retried to release the processor bus. In parallel with this, the remote access cache makes an exclusive copy of the required block. The reason is that making a room for the locked read may need to use the processor bus such that invalidation before writing back a shared block.

2.4.4.4 Replacement of RAC blocks

When a block of remote access cache has to be replaced to make a new room, the remote access cache may find the block in RAC_S or RAC_M state. If the state is RAC_S, the remote access cache can remove it without notification after evicting shared copies from processor caches. If the state is RAC_M, the remote access cache writes it back to home node after evicting copies from processor caches, as shown in figure 2.16.

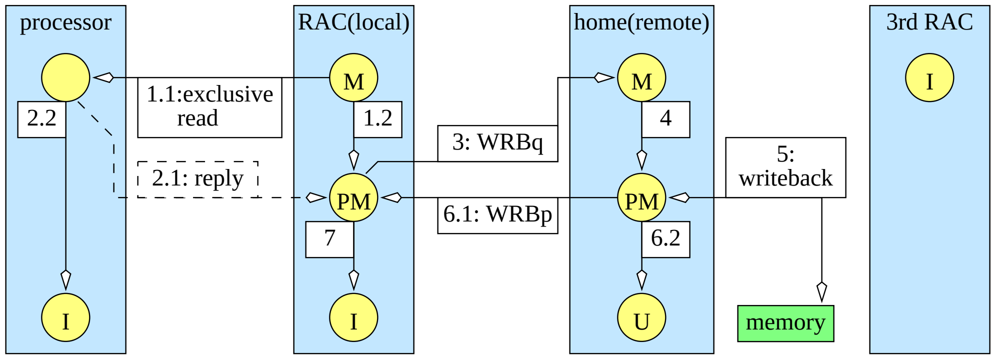


Figure 2.16: Remote access cache replacement

Figure 2.17 explains the ghost-sharing situation which is incurred after a remote access cache replaces its shared block without replacement-hint, the directory at the home node points to non-existing copy. Therefore any invalidation requests to nonresident blocks in the

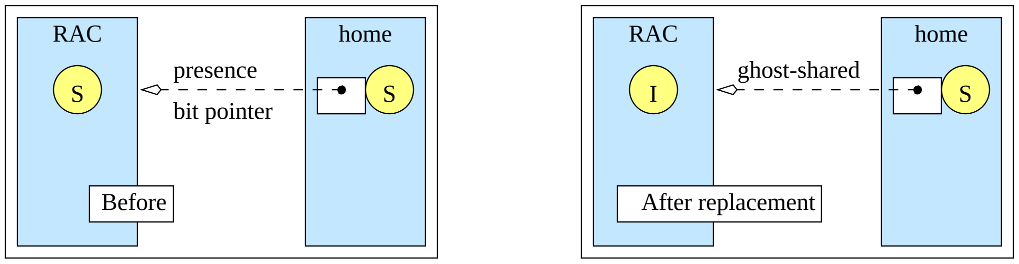


Figure 2.17: Ghost-sharing after replacement of SHARED block

remote access cache must be acknowledged, since the home node still has sharing link (bit map) to non-existing block due to un-notified replacement of shared blocks.

2.4.4.5 Concurrent accesses

Previous sections describes the RACE protocol under simple condition, where concurrent/parallel requests are absent. This section describes situations where requests having the same address can progress in parallel and how those concurrent requests are handled. This section first explains basic rules how to treat requests hitting on pending state and then describes the approach to guaranteeing forward progress. Since this protocol uses distributed state, it is possible that a message sent to another node can be incompatible with the state of the destination node when it arrives. To handle this kind of situation, the protocol simply replies with negative acknowledgement which makes the requesting node check its local state and to take an appropriate action again.

Because all requests are forwarded to the home node, an order of accesses can be established. Therefore, requests are severed in first-come-first-serve fashion. Requests for data hitting on pending state must retried. However, requests like invalidate and writeback are treated differently.

1. Request negative acknowledged except hitting on pending state must retries at once.
2. Request for data such as `CRDq` and `ERDq`, hitting on pending state must be negative acknowledged, which makes the requestor retry. If the requestor is directory, it retries the original request at once. If the requestor is remote access cache, it retries after a random amount or sufficient time delay.
3. Request for invalidation (*i.e.* `INVq`) hitting on pending state at remote access cache is acknowledged normally, so the requestor (*i.e.* directory) gets normal invalidation reply (*i.e.* `INVp`). Request for invalidation hitting on pending state at directory must be negative acknowledged, since something had happened before.
4. Writeback request (*i.e.* `WRBq`) hitting on pending state at directory must not be negative acknowledged instead it must updates the home memory at once,[3] since it carries most up-to-date data.

2.4.4.5.1 Multiple exclusive requests Figure 2.18 shows that how two exclusive remote requests to the same address arriving at the home are resolved. Assuming a situation where the 4th remote access cache contains the block in modified state and the 1st and the 2nd remote access caches want to acquire writable exclusive copies of it. Since the directory serializes requests to the same address, one of the exclusive requests is resolved first as indicated `1.1` to `5.2`. The other exclusive request is negative acknowledged since it hits on DIR_{PM} as numbered `a` and `b`. After receiving negative acknowledgement, the 1st remote access cache waits for a while and retires exclusive request again.

When multiple processors try to write a block at the same time, a livelock situation could arise. This livelock is resolved by letting the first request arriving at the home goes through, but making all the others retry by negative acknowledgments.

2.4.4.5.2 Multiple invalidation requests Figure 2.19 shows the case that two remote access caches containing the same block in shared state want to get writable exclusiveness.

[3] Practically, the directory holds the data and forwards it to the requesting node and then updates the home memory if necessary.

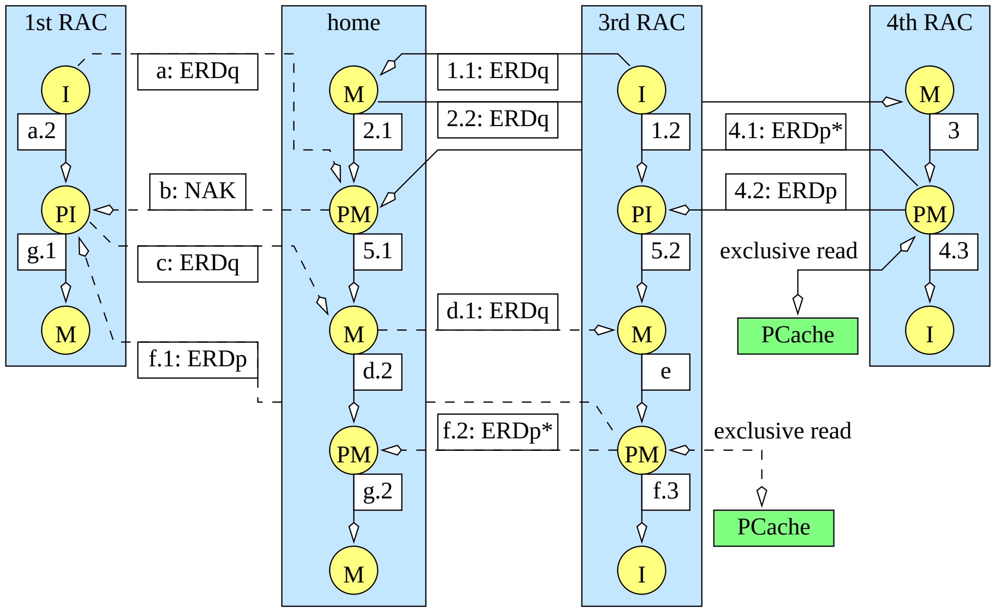


Figure 2.18: Concurrent exclusive remote reads from the 1st and 3rd RAC's: `4.1:ERDp*` and `f.2:ERDp*` does not carry data.

Like the previous example, the directory handles one of two invalidations first as indicates `1.1` to `5`. As it was mentioned before, the invalidation request hitting on pending state at the 1st remote access cache is replied by a normal invalidation reply as indicated `2.3` and `3.1`. When the second invalidate request from the 1st remote access cache is replied by a negative acknowledgement, the 1st remote access cache knows that it had be invalidated by previous invalidation from the directory, so it changes its request type to exclusive read instead of invalidation, since its cached copy is no longer valid. It should be noted that typically a remote access cache does not expect to receive data as a result of invalidation, since the block is in shared state. However, in the case of a race between invalidations from different agents (as shown in figure 2.19), agents may need to retrieve up-to-date data instead of invalidation. Only one (the first arriving invalidation) will win and the winning invalidation will not need a data transfer, but the others will need the supply the new modified data. Therefore, the processor bus protocol and cache protocol must have a mechanism to accept data in response to an invalidation (unexpected data reply mechanism). One more thing to be noted is that the remote access cache receiving data as a result of invalidation request (*e.g.*, the case of 1st RAC of Figure 2.19) needs to invalidate the processor cache lines in the processing node before providing the data if the remote access cache block contains more than one processor cache lines. The reason is that the data received as the result of invalidation is most up-to-date so that the data resides in the processor cache could be no longer valid.

2.4.4.5.3 Writeback hitting on pending state Figures 2.20 and 2.21 show the case where a data request arrives at the remote access cache in the middle of writeback. First see figure 2.20. Assuming that the 1st remote access cache issues a writeback as a result of replacement and the 3rd remote access cache wants to get an exclusive copy of the block. The exclusive request arrives at the directory and is forwarded to the 1st remote access cache since

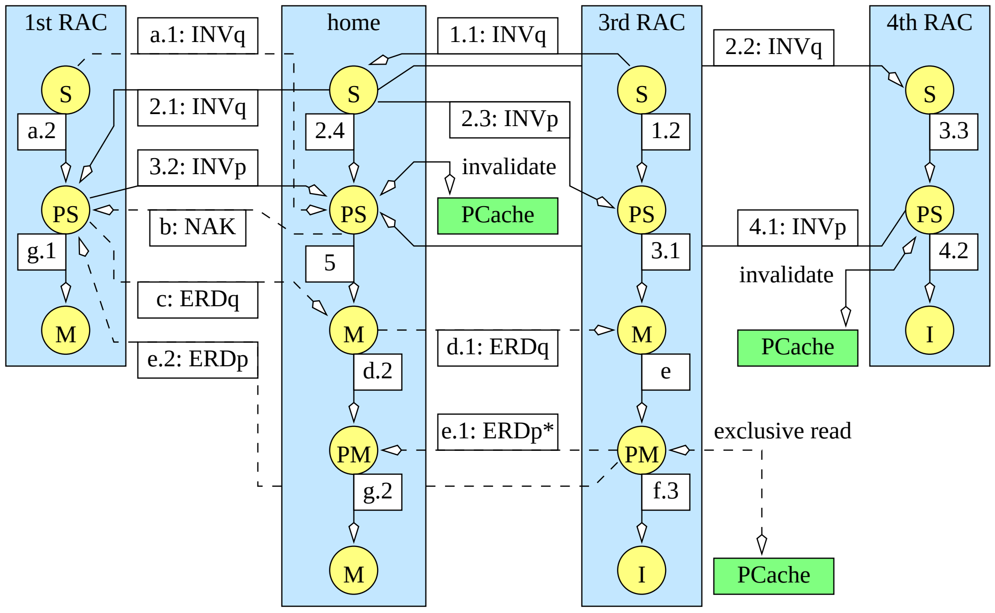


Figure 2.19: Concurrent invalidates from the 1st and 3rd RAC's: `e.1:ERDp*` does not carry data

the directory knows most up-to-date block resides in the 1st remote access cache. On arriving at the 1st remote access cache, the exclusive request hits on pending state, so it is negative acknowledged. Before the directory receives the negative acknowledgement, the writeback request arrives at the directory since the network guarantees in-order message delivery (see section 2.4.2). As it was mentioned before, the writeback request is performed although it hits on pending state at the directory. When the directory receives the negative acknowledgement, it knows that up-to-date block has been received, so it replies using the block. In the figure 2.20, the 3rd remote access cache wants to get non-exclusive copy, so that the home provides the data and then writes it back in to the memory.

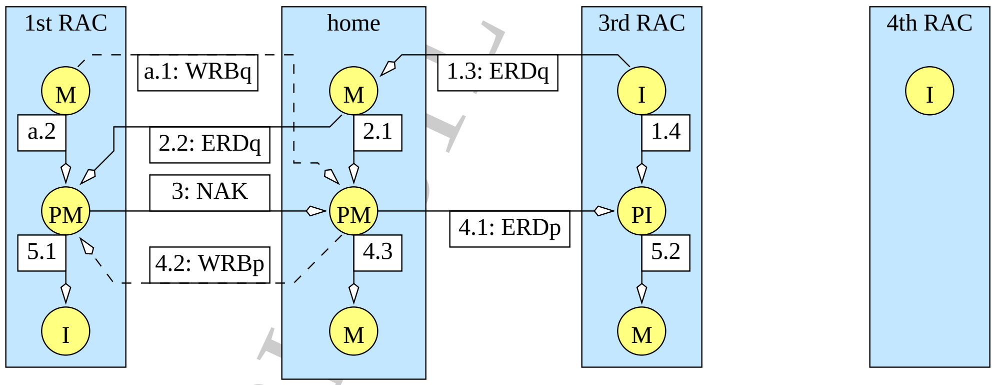


Figure 2.20: Concurrent writeback and exclusive read from the 1st and 3rd RAC's

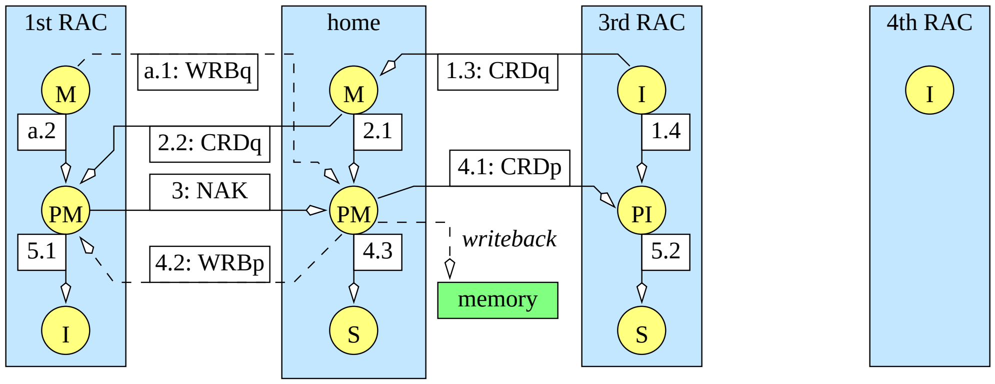


Figure 2.21: Concurrent writeback and coherent read from the 1st and 3rd RAC's

2.4.4.5.4 Multiple invalidates case 2 Figure 2.22 shows the case that two remote access caches containing the same block in shared state want to get writable exclusiveness and the processor at the home node also wants to invalidate other copies. The processor is not allowed to proceed its invalidate request unless it sees the state in DIR_S, since the processor's request hitting on pending state should be retried. Therefore the directory handles processor's invalidation first as indicates `a` to `e`. In the meantime, the 3rd and 4th remote access caches issue their invalidation requests to the home. Other invalidation requests hitting on pending state at the home are replied by negative acknowledgements, but the invalidation request by the home hitting on pending state at the 3rd and 4th remote access caches are replied by a normal invalidation reply as indicated `c.1` and `c.2`. When the 3rd and 4th remote access caches receive negative acknowledgements, they know that it had be invalidated by previous invalidation from the directory (home), so it changes its request type to exclusive read instead of invalidation, since its cached copy is no longer valid. After receiving negative acknowledgement, the remote access cache follows the sequence of exclusive read request. As mentioned before (see Figure 2.19), the 3rd and 4th remote access caches need to invalidate internal processor cache lines before providing the data received as the result of invalidation.

2.4.4.5.5 Writeback hitting on pending state case 2 Figure 2.23 shows the case that the local processor wants to get the data which resides in the remote access cache as DIR_M state. When the remote access cache writes back the data in the middle of request by the processor, the request by the directory hitting on pending state at the remote access cache is replied by a negative acknowledgement, but the writeback request hitting on pending state at the directory gets accepted as normal. When the directory receives the negative acknowledgement, it knows that up-to-date data has been received, so it provides the data to the processor and writes it back in to memory as well. If the processor wants to an exclusive copy, the directory just provides the data to the processor without writeback.

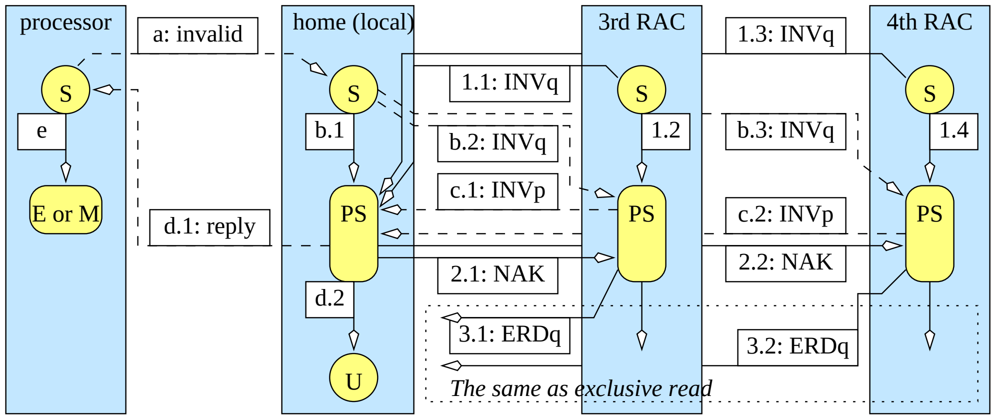


Figure 2.22: Concurrent invalidates from the local processor and the 1st and 3rd RAC's

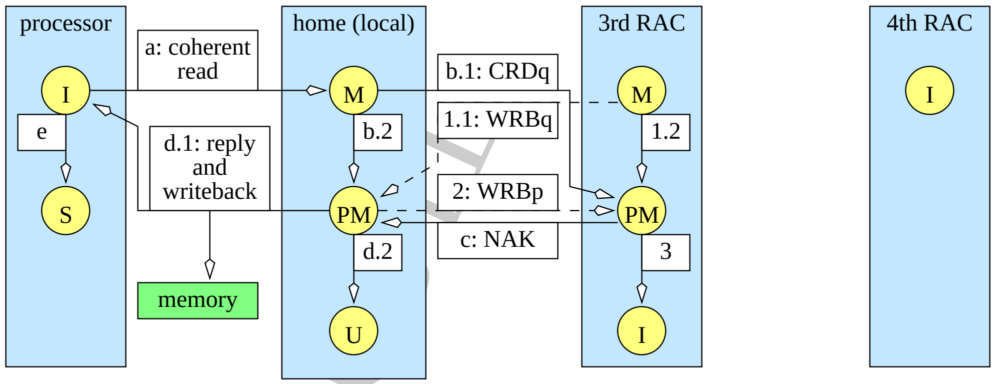


Figure 2.23: Concurrent writeback and coherent read from the processor

Chapter 3

An implementation of the RACE protocol for Intel SHV

3.1 Introduction

This chapter describes an implementation of the RACE protocol introduced in chapter 2 for Intel SHV (Standard High Volume).

3.2 Intel SHV

Intel SHV [5] is a symmetric multiprocessors containing 4 Pentium® II processors [2, 7] with private caches, memories, and I/O's, as shown in figure 3.1. Within a SHV, cache coherence is maintained using a typical MESI snooping cache protocol.

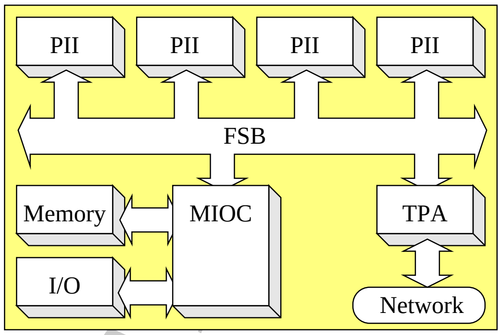


Figure 3.1: Intel Standard High Volume: Pentium® II Processor (PII); Memory and I/O Controller (MIOC); Front Side Bus (FSB); Third-Party Agent (TPA); Input/Output (I/O)

The MIOC (Memory and I/O Controller) [6] claims all memory and I/O requests mapped in the SHV, but the TPA (Third-Party Agent) resolves all remote accesses and enables the SHV to be interconnected via interconnection network. All bus agents including processors, MIOC, and TPA are connected through the Slot 2 processor bus called FSB (Front Side Bus) which is fully compliant to Pentium® Pro [3] bus.

In this work, it is assumed that the TPA consists of directory, remote access cache, and network interface.

3.3 A brief review of Pentium® Pro bus protocol

This section provides an overview of the Slot 2 processor bus called FSB in the context of data transfer protocol and transactions.

3.3.1 Six phases

The FSB utilizes Pentium® Pro processor [3] bus protocol, which is a synchronous, demultiplexed (separated address and data buses), and split protocol. Each Pentium® Pro processor bus transaction consists of up to six phases as shown in figure 3.2: *i.e.* arbitration, request, error, snoop, response, and data phases.

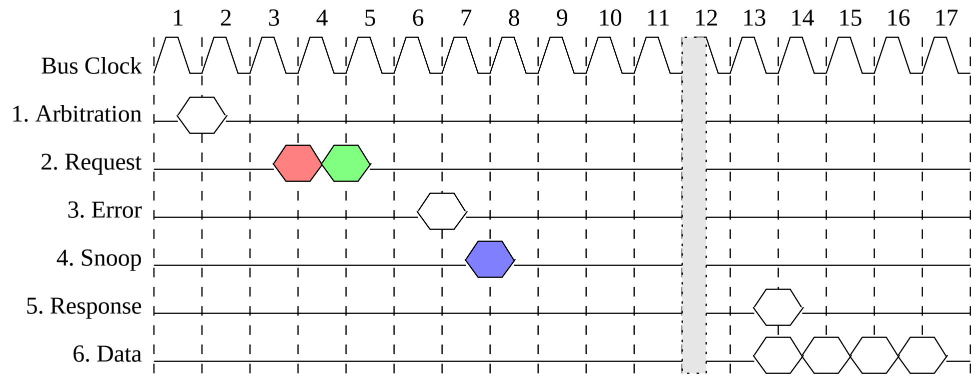


Figure 3.2: Pentium® Pro processor bus transaction phases: read a line case where phases 1 and 2 are performed by requestor, phases 3 and 4 by any bus agents, and phase 5 and 6 by responder.

1. ***Arbitration Phase***: The bus agent acquires an ownership of the bus to start a bus transaction. (`BREQ[3:0]`, `BPRI`, `BNR`, `LOCK`)

2. ***Request Phase***: The bus agent drives address and request information during two bus clocks. (`A[35:3]`, `REQ[4:0]`, `AP[1:0]`, `RP`, `ADS`)

3. ***Error Phase***: Any errors that occur during the request phase are reported in this phase. (`AERR`)

4. ***Snoop Phase***: All caching agents (or snooping agents) drive snoop signals to appropriate values in this phase to enforce cache coherency. (`HIT`, `HITM`, `DEFER`)

5. ***Response Phase***: The target device addressed during the request phase drives the transaction response during this phase. (`TRDY`, `RS[2:0]`, `RSP`)

6. ***Data Phase***: The target agent drives or accepts the transaction data, if there is any. (`D[63:0]`, `DEP[7:0]`, `DRDY`, `DBSY`)

It must be noted that the phases 1 to 4 must be kept its order. After the requesting agent becomes the bus owner, the transaction enters the request phase in which address and request information is driven on the bus. The error phase occurs three clocks after the request phases begins. The snoop phase occurs four or more clocks from the request phase. Every transaction without error during the error phase has a response phase which indicates whether the transaction has failed or succeeded, whether transaction completion is immediate or deferred, whether the transaction will be retried, and whether the transaction contains a data phase.

The Pentium® Pro processor bus architecture supports pipelined transactions in which multiple bus transactions can be overlapped. Each bus agent has In-order Queue (IOQ) which enables the bus agent to track every transaction that is issued to the bus. In addition to this, the IOQ guarantees in-order completion, since transactions receive their responses and data in the same order as they were issued. (see '3.3.2 Bus Transaction Pipelining and Transaction Tracking' in [3].)

3.3.2 Deferred transaction

The fact that I/O requests and remote accesses usually takes much longer time than local memory accesses, can make the bus under utilized, since other requests in the IOQ must wait until long delay requests are completed. To relieve this problem, the deferred transaction mechanism is provided in the Pentium® Pro processor bus. Using this mechanism, completion of a transaction can be deferred until the responding agent is ready to reply. To do this there is a signal called `DEFER` that can be driven during the snoop phase. Figure

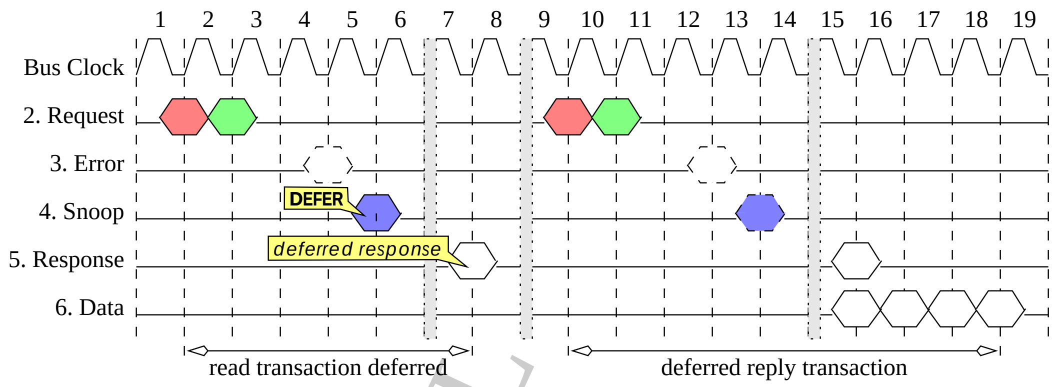


Figure 3.3: Pentium® Pro processor bus deferred transaction: Note that the arbitration phases are not shown and `DEFER` signal is driven during the snoop phase.

3.3 shows a typical deferred situation where a read transaction is deferred and is completed by the corresponding deferred reply transaction. More detailed description can be found in chapter 5 of [3].

It should be noted that the following transactions can not be deferred: transactions in a bus-locked operation, deferred reply transactions, and writeback transactions. Refer to 'section 5.3.3 Deferred Operations' in [3].

3.3.3 How the TPA is involved in the FSB activities

Since the TPA has directory and remote access cache, the TPA must snoop all bus memory transactions. The TPA must claim all remote accesses, so it acts like a memory. In addition to this, although a local accesses targeted to the local memory (*i.e.* MIOC), the TPA needs to intervene. For example the TPA knows that more recent data than the memory is cached in a remote node. By intervention, the TPA directs the MIOC to defer the transaction through `TPCTL` signals (shown in table 3.1) at the snoop phase and it also drives `DEFER` signal at the same time. On receiving `TPCTL` signals, the MIOC terminates the request by performing the deferred response, since the MIOC is always the owner of the transaction and hence must be the responding bus agent. After fetching the latest data from the remote node, the TPA performs the deferred reply transaction.

Table 3.1: Third-Party control signals (see [6])

Meaning	`TPCTL[1:0]`	action by MIOC
Accept	00	The MIOC accepts the request, and provides the normal response. The third-party agent is not involved in the transaction.
Hard Fail	01	Not supported.
Retry	10	The MIOC will generate a retry response. The access will be retried by the requesting agent.
Defer	11	The MIOC will issue a defer response, and the third-party agent will complete the transaction at a later time using a deferred reply. The third-party agent drives `DEFER` signal.

3.4 Pentium® Pro processor bus transaction types

3.4.1 Classification of Pentium® Pro bus transactions

Pentium® Pro [3] bus transactions can be divided into three classes: memory, I/O, and other as shown in table 3.2.

The memory transactions are used to transfer data to and from memory. The I/O transactions are used to transfer data to and from I/O address space. The other transactions perform special functions. The memory transactions are cache-coherent and require snooping, but the I/O and the other transactions are not snooped.

From data transfer's point of view, each transaction class can be divided into data transaction and non-data transaction. The data transactions are divided further into read-data, write-data, and deferred transactions. The read-data transactions expect to receive data. The write-data transactions provide data. The deferred transactions may or may not send data. The non-data transactions require no data transfer.

3.4.2 Memory types

In order to understand transfer types, *memory type* [3, 4] should be understood. Within individual pages or regions of system memory, Pentium® Pro memory system is allowed to

Table 3.2: Pentium® Pro bus transactions (see chapter 5 in [3])

		class		
		memory	I/O	other
data	read	memory read memory (read) invalidate line	I/O read	interrupt acknowledge
	write	memory write	I/O write	branch trace message
	deferred			deferred reply
non-data				shutdown flush halt sync flush acknowledge stop grant acknowledge SMI acknowledge

specify the types of caching as followed.

- **Uncacheable (UC)** – System memory locations are not cached. All reads and writes appear on the system bus and can have side effects. This type of cache control is useful for memory-mapped I/O devices.
- **Write Combining (WC)** – System memory locations are not cached and coherency is not enforced by the processor's bus coherency protocol. Reads cannot have side effects. All reads are not cached and writes may be delayed and combined in the write buffer. This type of cache control is appropriate for frame buffers, where the order of writes is unimportant as long as the writes update memory.
- **Write Through (WT)** – Reads come from cache lines on cache hits; read misses cause cache fills. All writes are written to a cache line (when possible) and through to system memory. When writing through to memory, invalid cache lines are never filled, and valid cache lines are either updated or invalidated.
- **Write Protected (WP)** – Reads come from cache lines when possible, and read misses cause cache fills, while writes bypass the cache entirely. Writes are propagated to the system bus and cause corresponding cache lines on all processors on the bus to be invalidated. This type only protects lines in the cache from being updated by write, but does not protect main memory.
- **Write Back (WB)** – Writes and reads to and from system memory are cached. Reads come from cache lines on cache hits; read misses cause cache fills. Write misses cause cache line fills, and writes are performed entirely in the cache, when possible.

The Pentium® Pro cache protocol assumes that different caching agents (different Pentium® Pro processors) agree on the memory type and cache attributes of each memory line (see chapter 6 of [3]). Memory type range register updates to change memory type are permitted only if all caches have been flushed before and after the update.

Table 3.3: Memory types and their properties (see chapter 11 of [4])

Memory type	ATTR[2:0] Ab[16:24]	cacheable in L1 and L2		writeback cacheable
		read	write	
uncacheable	000	NO	NO	NO
write combining	100	NO	NO	NO
write through	101	YES	YES/NO[1]	NO
write protected	110	YES	NO	NO
write back	111	YES	YES	YES

Notes: 1) Valid lines may be updated, while invalid lines are never filled.

Table 3.3 shows the memory types that can be specified and their properties. L3 cache agents (*i.e.* TPA) use the memory type attributes to determine cache line allocation policy. On read misses, L3 cache fills cache if memory type is one of write back, write through, and write protected. On write misses, L3 cache allocates line only if memory type is write back.

3.4.3 Transfer types

Among six phases of a bus transaction, the *request phase* carries the *request information* and consists of two clocks long. During the phases, `REQ[4:0]` and `A[35:3]` that are called *request signals* are driven to transfer the request information. Since the request signals are driven two times during the request phase, letters 'a' and 'b' are used to distinguish which clock. For example, `REQa` signal is driven at the first clock and `REQb` at the following clock. In addition to the request signals, `LOCK` signal is needed to classify transfer types.

The memory type specified by `ATTR` determines detailed meaning of transaction. Table 3.4 shows decoding (see chapter 3 of [3]).

Table 3.4: Pentium® Pro bus transfer types

class	inputs					decoded types
	LOCK	REQa	REQb[1:0]	Ab[15:8]	Ab[26:24]	
			LEN[1:0]	BE[7:0]	ATTR[2:0]	
other	0	00000	xx	XX	xxx	deferred reply
	0	01000	00	XX	xxx	interrupt acknowledge
	0	01000	01	01	xxx	shutdown
	0	01000	01	02	xxx	flush
	0	01000	01	03	xxx	halt
	0	01000	01	04	xxx	sync
	0	01000	01	05	xxx	flush acknowledge
	0	01000	01	06	xxx	stop grant ack.
	0	01000	01	07	xxx	SMI acknowledge
	0	01001	00	XX	xxx	branch trace message
I/O	0	10000	00	XX	xxx	I/O read

continued on next page

continued from preview page (Table 3.4)

class	inputs					decoded types
	`LOCK`	`REQa`	`REQb[1:0]`	`Ab[15:8]`	`Ab[26:24]`	
			`LEN[1:0]`	`BE[7:0]`	`ATTR[2:0]`	
	0	10001	00	XX	xxx	I/O write
	0	xx010	10	FF	$\overline{000}$[2]	read invalidate line
	0	xx010	00	00	$\overline{000}$[2]	invalidate line
	0	xx1x0	10	FF	x$\overline{00}$[3]	read line
	0	xx1x0	01	FF	x$\overline{00}$[3]	read part-line
	0	xx1x0	00	XX[4]	x$\overline{00}$[3]	partial read
	0	xx1x0	10	FF	x00[5]	uncached read line
	0	xx1x0	01	FF	x00[5]	uncached read part-line
memory	0	xx1x0	00	XX[4]	x00[5]	uncached partial read
	0	xx101	10	FF	111	write line[8]
	0	xx101	01	FF	111	write part-line
	0	xx101	00	XX[4]	111	partial write
	0	xx1x1	10	FF	x00[5]	uncached write line
	0	xx1x1	01	FF	x00[5]	uncached write part-line
	0	xx1x1	00	XX[4]	x00[5]	uncached partial write
	0	xx111	10	FF	1$\overline{00}$[6]	write through line[9]
	0	xx111	01	FF	1$\overline{00}$[6]	write through part-line
	0	xx111	00	XX[4]	1$\overline{00}$[6]	partial write through
	0	xxxx0	01	00	xxx	NULL[7]
	1	xxxx0	00	XX	xxx	locked read
	1	xxxx1	00	XX	xxx	locked write

Notes: 1) 'x' and 'X' imply don't care for bit and nibble, respectively.
2) Any combination except 000. (WB, WP, WT, WC)
3) Any combination except 00. (WB, WP, WT)
4) Active bits indicate valid bytes.
5) One of UC and WC memory types.
6) One of WP and WT memory types.
7) Due to transaction between memory and I/O device. TPA ignores it.
8) When `REQa[1]` is deasserted, no agent may assert `DEFER` to retry the transaction.
9) `REQa[1]` asserted indicates that the write transaction may be retried.

The memory class transaction can be further divided into the following.

- *cacheable*
  - **BRL** (Bus Read Line): This transaction is a memory read transaction for a full cache line and indicates that a requesting agent has had a read miss and does not intend to modify this line when the line is returned.
  - **BRIL** (Bus Read Invalidate Line): This transaction is a memory read transaction for a full cache line and indicates that a requesting agent has had a read miss and intends to modify this line when the line is returned.

- **BIL** (Bus Invalidate Line): This transaction is a memory read transaction for a 0 byte and and indicates that a requesting agent contains the line and intends to modify the line.
- **BWL** (Bus Write Line): This transaction indicates that a requesting agent issued a memory write transaction for a full cache line and implies that a requesting agent intends to write back a modified line or an I/O agent intends to write a line to memory.
- **BRP** (Bus Read Part-line): This transaction indicates that a requesting agent issued a memory read transaction for less than a full cache line. It moves a quantity of data smaller than a cache line but an even multiple of the chunk size (8Byte=64bit): *i.e.* 16Byte aligned transfer.
- **BWP** (Bus Write Part-line): This transaction indicates that a requesting agent issued a memory write transaction for less than a full cache line. It moves a quantity of data smaller than a cache line but an even multiple of the chunk size (8Byte=64bit): *i.e.* 16Byte aligned transfer.
- **BPR** (Bus Partial Read): This transaction indicates that a requesting agent issued a memory read transaction for less than or equal to 8Byte within an aligned 8Byte span.
- **BPW** (Bus Partial Write): This transaction indicates that a requesting agent issued a memory write transaction for less than or equal to 8Byte within an aligned 8Byte span.
- **BWTL** (Bus Write-Through Line):
- **BWTP** (Bus Write-Through Part-Line):
- **BPWT** (Bus Partial Write-Through):

• *locked*

- **BLR** (Bus Locked Read): This transaction indicates that a requesting agent issued a bus locked memory read transaction.
- **BLW** (Bus Locked Write): This transaction indicates that a requesting agent issued a bus locked memory write transaction.

• *Uncached*

- **BURL** (Bus Uncached Read Line):
- **BURP** (Bus Uncached Read Part-Line):
- **BUPR** (Bus Uncached Partial Read):
- **BUWL** (Bus Uncached Write Line):
- **BUWP** (Bus Uncached Write Part-Line):
- **BUPW** (Bus Uncached Partial Write):

• *Input/Output*

- **BIOR** (Bus Input/Output Read): This transaction indicates that a requesting agent issued a I/O-mapped read transaction.

- **BIOW** (Bus Input/Output Write): This transaction indicates that a requesting agent issued a I/O-mapped write transaction.

Table 3.5 shows a grouping of transactions. Since, according to Pentium® Pro and MIOC

Table 3.5: Pentium® Pro bus transactions

	read (get)		write (put)	
cacheable	BRIL BIL	Read Invalidate Line Invalidate Line		
	BRL BRP BPR	Read Line Read Part-line* Partial Read*	BWL BWP BPW	Write Line Write Part-line* Partial Write*
			BWTL BWTP BPWT	Write-Through Line Write-Through Part-line* Partial Write-Through
uncached	BURL BURP BUPR	Uncached Read Line* Uncached Read Part-line* Uncached Partial Read	BUWL BUWP BUPW	Uncached Write Line* Uncached Write Part-line* Uncached Partial Write
Locked	BLR	Locked Read	BLW	Locked Write
I/O	BIOR	I/O Read	BIOW	I/O Write
special				

Remarks: 1) Transactions with * may not occur on the Slot 2 processor bus (FSB).

specifications, both do not generate part-line transactions, and the Pentium® II processor with cache does not generate partial accesses to cacheable address space, some of listed transaction may not occur.

3.5 Pentium® Pro cache protocol

3.5.1 Cache line states

The Pentium® Pro processor cache protocol belongs to a family of cache protocol called *MESI protocols*. Each line which is the unit of caching, can be one of the following states.

- **Invalid** (I): The line is not available in this cache. An access to this line misses the cache and can cause the processor to fetch the line into the cache from the bus (from memory or from another cache).
- **Shared** (S): The line is in this cache, contains the same value as in memory, and can have the Shared state in other caches. Processor's reading the line causes no bus activity. Processor's writing the line causes an invalidate transaction to gain ownership of the line.
- **Exclusive** (E): The line is in this cache, contains the same value as in memory, and is Invalid in all other caches. Processor's reading the line causes no bus activity. Processor's writing the line causes no bus activity, but changes the line's state to Modified.
- **Modified** (M): The line is in this cache, contains a more recent value than memory, and is Invalid in all other caches. Processor's reading or writing the line causes no bus activity.

3.5.2 Coherency-related signals

All caching agents on the bus are required to snoop all bus memory transactions and uses two signals, `HIT` and `HITM`, to indicate the snoop result during the snoop phase.

- `HIT`: The snooping agent contains a shared copy of the cache line and driving this signal indicates that the snooping agent intends to retain the line in Shared state in its cache after the snoop phase.
- `HITM`: The snooping agent contains a modified copy of the cache line and driving this signal informs that the asserting agent will supply the line instead of memory. It prevents memory from suppling the line and makes the memory accept the line to update it in the memory (*i.e.* implicit writeback).

There is one more signal called `DEFER`, which can be driven during the snoop phase. An activation of the signal indicates intention to defer completion of a transaction and cancelation of the transaction from IOQ. `DEFER` overrides `HIT`, but cannot override `HITM`. When `HITM` and `DEFER` are both active during the snoop phase, `HITM` is given priority and the transaction must be completed with implicit writeback response.

If both `HIT` and `HITM` are driven together in the snoop phase, it means that a caching agent is not ready to indicate snoop status, and it needs to stall the snoop phase. The snoop signals including `DEFER` are sampled again two clocks later. This process continues as long as the stall state is sampled. (see '4.4.2.2. Stalled Snoop Phase' in [3].)

3.6 Two steps approach for snooping

From implementor's perspective, two steps approach would be needed for snoop logic to drive snoop signals at the snoop phase, as shown in figure 3.4. This approach enables the snoop

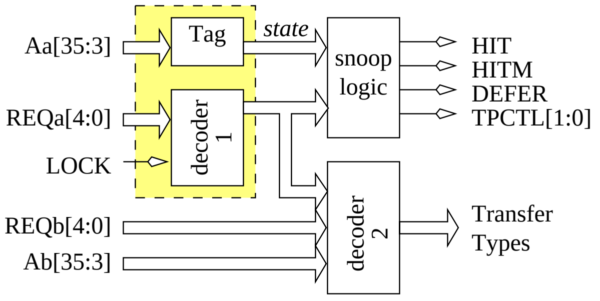


Figure 3.4: Two steps approach to get transfer information: shaded part indicates the 1st step.

logic to have possibly sufficient time to decide which snoop signals are going to be driven. At the first step, the requesting signals driven at the first clock of the requesting phase are used to get transfer type information, that is utilized by bus snooping logic. At the second step, the transfer type information acquired at the first clock and the requesting signals driven at the second clock are used to get precise transfer type information. For the second step, refer to section 3.4.3.

3.6.1 The first and second steps

The first step only uses `REQa` and `LOCK` signals and table 3.6 shows its decoding.

Some bits of `REQa` have special meanings. `REQa[0]` indicates read and write such that write when active. `REQa[2:1]` indicates memory transaction class so that transactions must be snooped when both or one of `REQa[2]` and `REQa[1]` are active. As a result, snoop logic at figure 3.4 gets only accept the memory class transactions and doesn't care of the I/O and the other classes.

The second step uses the first step signals and `REQb` and `Ab` signals. During this step, bus transfer types are decoded as discussed in section 3.4.

3.6.2 Snooping tables

Snoop logic must ignore I/O and other transaction classes and, in addition, memory transactions with `REQa[4]` = 1. Therefore the following combination can be used as a snoop enable indication: `(REQa[4]=0)&&((REQa[2]=1)||(REQa[1]=1))`, where the first item selects 32 or 36 bits addressing and the last two items select memory transactions.

Table 3.7 shows how the snoop signals are generated for local accesses. It is noted that the MIOC always the owner of transactions to local, so the MIOC must performs response at the response phase. When it is needed that the TPA intervenes ongoing transaction, the TPA drives `DEFER` signal and directs the MIOC to perform the deferred or retry response through `TPCTL` at the same time. When the TPA directs the MIOC to perform deferred response, it

Table 3.6: The 1st step decoding table

class	inputs						decoded types
	LOCK	REQa[4:0]					
		4	3	2	1	0	
other	0	0	0	0	0	0	deferred-reply
	0	0	1	0	0	0	interrupt-special
	0	0	1	0	0	1	branch-trace
I/O	0	1	0	0	0	0	I/O-read
	0	1	0	0	0	1	I/O-write
memory	0	x	x	0	1	0	exclusive-read
	0	x	x	1	x	0	coherent-read
	0	x	x	1	0	1	writeback
	0	x	x	1	1	1	write through
	1	x	x	1*	1*	0	locked-read
	1	x	x	1*	1*	1	locked-write

Notes: 1) 'x' implies don't care.
2) At least one bit of the bits indicated * must be active.

performs the deferred transaction to complete the original transaction after the TPA doing all necessary actions. Note that I/O-read and I/O-write are note considered.

Table 3.7: Snooping table for local accesses

bus types	DIR state	signals at the snoop phase				next DIR	remarks
		HIT	HITM	DEFER	TPCTL		
coherent read	U	0	0	0	00	U	
	S	1	0	0	00	S	
	M	0	0	1	11	PM	MIOC performs deferred response
	PU	0	0	1	10	PU	MIOC performs retry response
	PS	0	0	1	10	PS	MIOC performs retry response
	PM	0	0	1	10	PM	MIOC performs retry response
exclusive read	U	0	0	0	00	U	
	S	0	0	1	11	PS	MIOC performs deferred response
	M	0	0	1	11	PM	MIOC performs deferred response
	PU	0	0	1	10	PU	MIOC performs retry response
	PS	0	0	1	10	PS	MIOC performs retry response
	PM	0	0	1	10	PM	MIOC performs retry response
write back	U	0	0	0	00	U	
	S	0	0	0	00	S	may not occur
	M	0	0	0	00	M	may not occur
	PU	0	0	0	00	PU	
	PS	0	0	0	00	PS	may not occur
	PM	0	0	0	00	PM	may not occur

continued on next page

continued from previous page (Table 3.7)

bus types	DIR state	signals at the snoop phase				next DIR	remarks
		`HIT`	`HITM`	`DEFER`	`TPCTL`		
write through	U	0	0	0	00	U	
	S	0	0	1	11	PS	MIOC performs deferred response
	M	0	0	1	11	PM	MIOC performs deferred response
	PU	0	0	1	10	PU	MIOC performs retry response
	PS	0	0	1	10	PS	MIOC performs retry response
	PM	0	0	1	10	PM	MIOC performs retry response
locked read	U	0	0	0	00	U	
	S	0	0	1	10	PS	MIOC performs retry response
	M	0	0	1	10	PM	MIOC performs retry response
	PU	0	0	1	10	PU	MIOC performs retry response
	PS	0	0	1	10	PS	MIOC performs retry response
	PM	0	0	1	10	PM	MIOC performs retry response
locked write	U	0	0	0	00	U	
	S	-	-	-	–	S	must not occur
	M	-	-	-	–	M	must not occur
	PU	0	0	0	00	PU	1)
	PS	0	0	0	00	PS	must not occur
	PM	0	0	0	00	PM	must not occur

Remarks: 1) Due to network-side accesses, but it is safe under bus holding mechanism.

Table 3.8 shows how the snoop signals are generated for remote accesses. It is noted that `TPCTL` signals are not used here, since the MIOC does not claim the remote accesses. It should be also noted that the 'deferred response' at the response phase is only allowed when `DEFER` (with `HITM` inactive) is asserted during the snoop phase. (see 'section 4.5.3.4 RS[2:0]# encoding' in [3].)

Table 3.8: Snooping table for remote accesses

bus types	RAC state	signals at the snoop phase			next RAC	remarks
		`HIT`	`HITM`	`DEFER`		
coherent read	I	0	0	1	PX†	TPA performs deferred response
	S	1	0	1	PS	TPA performs deferred response
	M	0	0	1	PM	TPA performs deferred response
	M	0	1*	1	PM	TPA takes implicit writeback
	L	-	-	-	L	must not occur
	PI	0	0	1	PI	TPA performs retry response
	PS	0	0	1	PS	TPA performs retry response
	PM	0	0	1	PM	TPA performs retry response
	PM	0	1*	1	PM	TPA takes implicit writeback
	PL	-	-	-	PL	must not occur
exclusive read	I	0	0	1	PX	TPA performs deferred response
	S	0	0	1	PS	TPA performs deferred response
	M	0	0	1	PM	TPA performs deferred response

continued on next page

continued from preview page (Table 3.8)

bus types	RAC state	signals at the snoop phase			next RAC	remarks
		HIT	HITM	DEFER		
	M	0	1*	1	PM	TPA takes implicit writeback
	L	-	-	-	L	must not occur
	PI	0	0	1	PI	TPA performs retry response
	PS	0	0	1	PS	TPA performs retry response
	PM	0	0	1	PM	TPA performs retry response
	PM	0	1*	1	PM	TPA may or may not take implicit writeback
	PL	-	-	-	PL	must not occur
write back	I	0	0	0	I	may not occur
	S	0	0	0	PS	may not occur
	M	0	0	0	PM	
	L	-	-	-	L	must not occur
	PI	0	0	0	PI	may not occur
	PS	0	0	0	PS	may not occur
	PM	0	0	0	PM	TPA handles the data carefully
	PL	-	-	-	PL	must not occur
write through	I	0	0	1	PX	TPA performs deferred response
	S	0	0	1	PS	TPA performs deferred response
	M	0	0	1	PM	TPA performs deferred response
	L	-	-	-	L	must not occur
	PI	0	0	1	PI	TPA performs retry response
	PS	0	0	1	PS	TPA performs retry response
	PM	0	0	1	PM	TPA performs retry response
	PL	-	-	-	PL	must not occur
locked read	I	0	0	1	PX	TPA performs retry response
	S	0	0	1	PS	TPA performs retry response
	M	0	0	1	PM	TPA performs retry response
	M	0	1*	1	PM	TPA takes implicit writeback
	L	0	0	0	L	
	PI	0	0	1	PI	TPA performs retry response
	PS	0	0	1	PS	TPA performs retry response
	PM	0	0	1	PM	TPA performs retry response
	PM	0	1*	1	PM	TPA may or may not take implicit writeback
	PL	0	0	0	PL	must not occur
locked write	I	-	-	-	I	must not occur
	S	-	-	-	S	must not occur
	M	-	-	-	M	must not occur
	L	0	0	0	L	
	PI	-	-	-	PI	must not occur
	PS	-	-	-	PS	must not occur
	PM	-	-	-	PM	must not occur
	PL	-	-	-	PL	must not occur

continued on next page

continued from preview page (Table 3.8)

bus types	RAC state	signals at the snoop phase			next RAC	remarks
		HIT	HITM	DEFER		

Notes: * HITM driven by other processor cache. [1]
† one of PI, PS, or PM.

3.7 Constraints

- There are no part-line aligned transfers on the FSB, since both the Pentium® Pro processor and the MIOC do not generate it.
- There are no I/O transactions targeted to or through the TPA. According to the MIOC specification, all I/O class transactions are claimed by the MIOC.
- It is assumed that split-lock operations do not occurs. As a result, it is not supported that a lock variable crosses the cache line boundaries since this causes split-lock operation. The most desirable one is a lock variable less than or equal to 8-byte and aligned 8-byte boundaries.
- As the Pentium® Pro cache protocol assumes that different caching agents agree on the memory type and cache attributes of each cache memory line, uncached accesses never hit on valid cache states, *e.g.*, DIR_S, DIR_M, RAC_S, and RAC_M.

[1]When HITM and DEFER are both active during the snoop phase, HITM is given priority and the transaction must be completed with implicit writeback response.

3.8 The RACE protocol for Intel SHV

In this section, the RACE protocol is explained in the context of Intel Standard High Volume (SHV).

This section consists of two sub-sections; one is about local accesses and the other for remote accesses. Each sub-section explains each transactions incurred by the processor, which are in table 3.5 in section 3.4.

3.8.1 Local memory operations

In this subsection, the cases in which requesting node is home node are explained. Therefore, all accesses dealt with in here normally are resolved by the local memory. However, the directory must snoop these local memory operations in order for the local memory not to provide stale data.

3.8.1.1 Coherent read cases (BRL)

BRL local request to DIR_U block reads data from local memory and untouches the directory.

BRL local request to DIR_S block reads data from local memory and untouches the directory. At snoop phase, the directory must assert `HIT`. By doing that, the processor cache changes its state not to PC_E but to PC_S.

BRL local request to DIR_M block makes the directory controller send a fetch request (CRD) to the owning node and change the state to DIR_{PM}. At the same time, the directory controller makes the processor wait for deferred reply by asserting `DEFER`[2]. The remote access

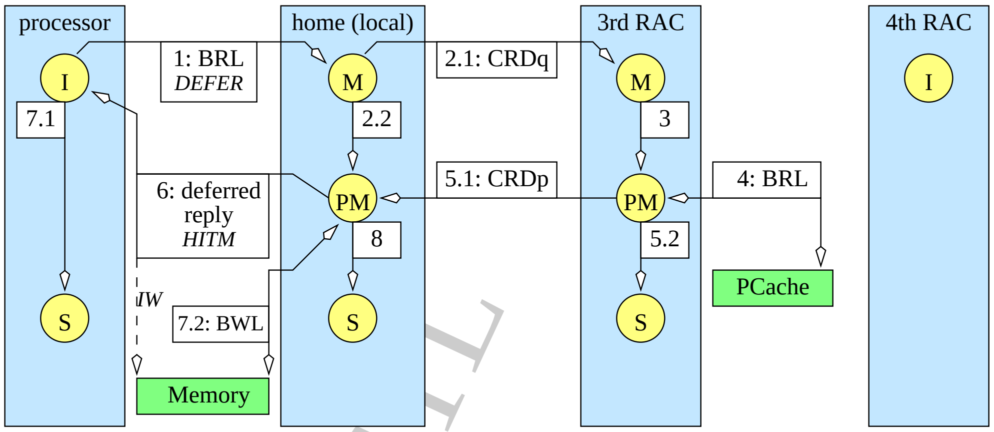


Figure 3.5: BRL hits on DIR_M: *IW* stands for 'implicit write back'.

cache at the owning node performs BRL on its processor bus to get up-to-date data, which also makes processors states be PC_S. Note, however, that if there is no `HITM` asserted by processor cache, the BRL must be terminated by the remote access cache performing a dummy response phase. Thereafter the owning node sends the data to the home node and goes to RAC_S.

[2]When directory controller (TPA) sees an access on the processor bus targeted to the main memory (MIOC) and the TPA knows that the most current data is cached in a remote location, the TPA drives `DEFER` signal on the bus and directs the MIOC to respond defer respond using `TPCTL` signals, so that latest data can be supplied via defer reply after fetching it from the remote location.

The directory controller responds using deferred reply and asserts **HITM** at snoop phase and provides the data that is also written back implicitly to the local memory. It has to be noted that the host processor expects critical-word-first order during implicit writeback. In addition to these, the directory controller writes back the remaining lines through BWL's and changes its state DIR_S as shown in figure 3.5. This approach has an advantage of critical-line-first order.

3.8.1.2 Exclusive read cases (BRIL)

BRIL local request to DIR_U block reads data from local memory and untouches the directory.

BRIL local request to DIR_S block makes the directory controller send invalidate requests (INVq's) to sharing nodes and change the state to DIR_{PS}. At the same time, the processor cache reads data from local memory. The directory controller changes the state to DIR_U after receiving all invalidate acknowledgements. The local memory provides the data to the processor cache as shown in figure 3.6. On receiving an invalidate request (INVq), the

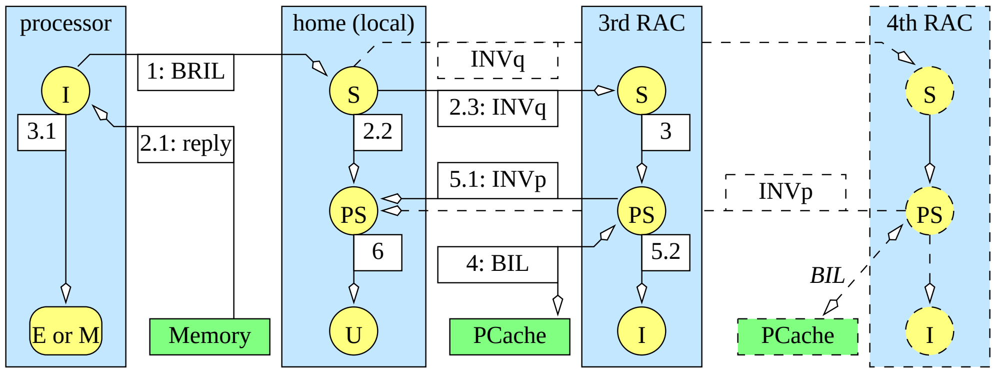


Figure 3.6: BRIL hits on DIR_S

remote access cache performs BIL on its processor bus, which invalidates the processor cache. Since none will response, the remote access cache must terminates the invalidate request by performing response.

BRIL local request to DIR_M block makes the directory controller send an exclusive fetch request (ERD) to the owning node and change the state to DIR_{PM}. At the same time, the directory controller makes the processor wait for deferred reply by asserting **DEFER**. The remote access cache at the owning node performs BRIL on its processor bus to get up-to-date data, which also makes processors states be invalidated. Note, however, that if there is no **HITM** asserted by processor cache, the BRL must be terminated by the remote access cache performing a dummy response phase. Thereafter the owning node sends the data to the home node and goes to RAC_I. The directory controller responds using deferred reply and asserts **HITM** at snoop phase and provides the data that is also written back implicitly to the local memory. In addition to these, the directory controller writes back the remaining lines through BWL's and changes its state DIR_U as shown in figure 3.7.

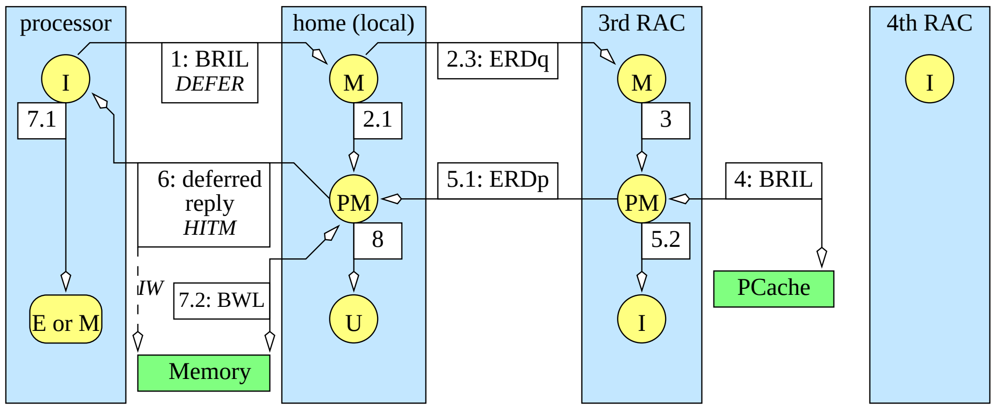


Figure 3.7: BRIL hits on DIR_M

3.8.1.3 Invalidate cases (BIL)

BIL local request to DIR_U block does not affect the directory and the processor bus operation resolves the request. BIL local request to DIR_M block does not occur since DIR_M means that a remote node has the block exclusively.

BIL local request to DIR_S makes the directory controller send invalidate requests (INVq's) to sharing nodes and change the state to DIR_{PS}. At the same time other processor cache at the same node are invalidated. The directory controller changes the state to DIR_U after receiving all invalidate acknowledgements. The requesting processor's cache gets exclusiveness of the data after the directory controller sends all invalidate requests as shown in figure 3.8. On receiving an invalidate request (INVq), the remote access cache performs BIL on its

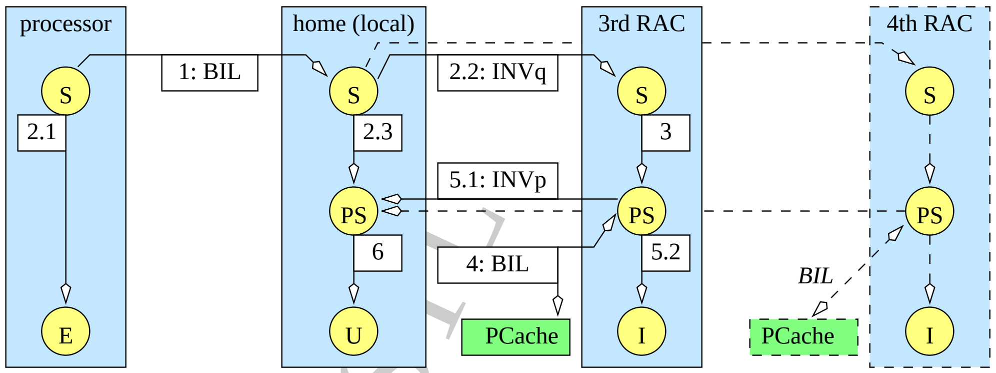


Figure 3.8: BIL hits on DIR_S

processor bus, which make the processor cache be invalidated. Since none will response, the remote access cache must terminates the invalidate request by performing response.

Typically the processor does not expect to receive data in response to the BIL, since the line may be in 'share' state in its cache. However, in the case of a race between BIL's from different processing nodes, it may need to accept up-to-date data as a result of BIL (unexpected data reply mechanism). For details, see section 2.4.4.5.

If the third-party agent defers an invalidate request (BIL), and subsequently wishes to provide data with the deferred reply, then the third-party agent must assert `HITM` during the deferred reply with data. This instructs the MIOC to look for and accept data arriving with the reply. [3]

3.8.1.4 Writeback cases (BWL)

BWL local request to DIR_S or DIR_M block does not occur, [4] since the processor generating BWL must have modified cache line (PC_M) meaning that there is no sharing. As a result, the directory needs not to care of BWL local request.

3.8.1.5 Write-through cases (BWTL and BPWT)

BWTL and BPWT local request to DIR_U writes data to the local memory and untouches the directory. BWTL and BPWT local request to DIR_S is handled as the same as a BIL local request, as shown in figure 3.9.

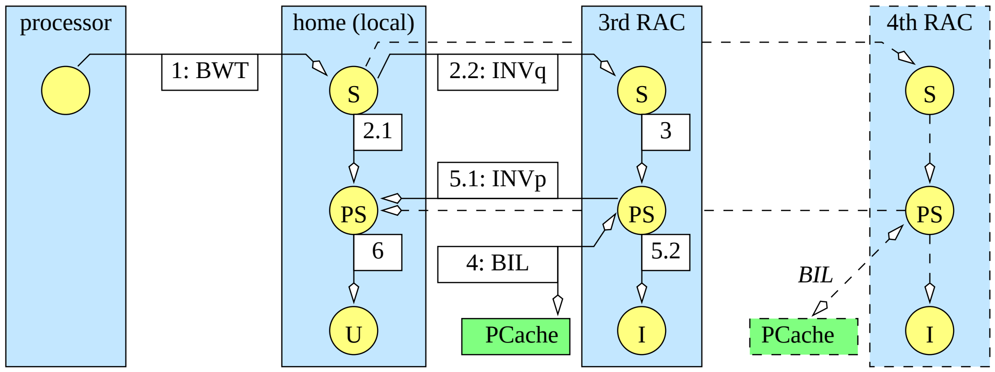


Figure 3.9: BWTL/BPWT hits on DIR_S

A BWRL or BPWT local request to DIR_M is handled as the same as an exclusive read and, in addition, the memory must merge the data written through and the data gotten by the exclusive read, [5] as shown in figure 3.10.

3.8.1.6 Uncached read/write cases (BUPR and BUPW)

BUPR/BUPW local request to DIR_S and DIR_M block does not occur, since DIR_S and DIR_M mean that some caching agents accessed the block as cacheable region. As a result, the directory needs not to care of BUPR/BUPW local request.

[3]See '5.2.4.1 Request Initiator Responsibilities (Deferring Agent)' in [3]. For a Deferred Reply resulting a Memory Invalidate Transaction which hit a modified line on another bus, the deferring agent must echo the `HITM` in the Snoop Result Phase of the Deferred Reply (the Snoop Result Phase indicates all changes in the length of data returned).
See '5.3.1.2 Requesting Agent Responsibilities' in [3]. If the original Invalidate Line Transaction receives a Deferred Reply, a `HITM` in the Snoop Phase indicates data will return, and the requesting agent updates its internal cache with the data.

[4]If memory write transaction with `REQa[1]` asserted hits a valid line in a snooping cache, a cache coherency violation has occurred. See '5.2.1.2 Memory Write Transactions' in [3].

[5]When writing through the cache, `REQa[1]` is asserted. This transaction is snooped and can receive an implicit writeback response. See '5.2.1.2 Memory Write Transactions' in [3].

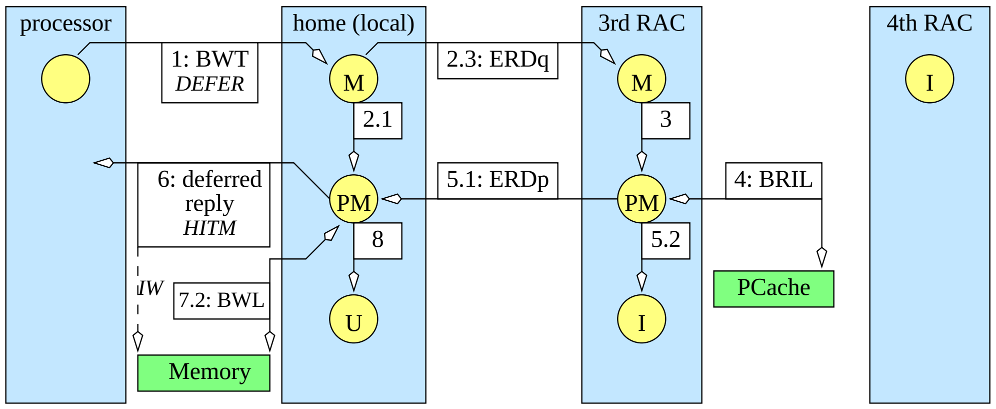


Figure 3.10: BWTL/BPWT hits on DIR_M

3.8.1.7 Locked read and write cases (BLR and BLW)

BLR local request to DIR_U block reads data from local memory and untouches the directory. The local memory interlocks the location until BLW is performed. By allowing BLR for DIR_U state, the processor bus will guarantee atomic read-modified-write operation between BLR and its corresponding BLW using bus lock mechanism.

BLR local request to DIR_S makes the processor retry[6]. (The directory asserts DEFER and $\texttt{TPCTL}_{retry}$ to abort the bus lock sequence.) The directory controller sends invalidate requests to sharing nodes and changes the state to DIR_{PS}, and then the directory controller changes the state to DIR_U after receiving all invalidate acknowledgements as shown in figure 3.11. BLR is retried until the directory gets the block in DIR_U state. After that, the remains of operation is the same as BLR hits on DIR_U block.

BLR local request to DIR_M makes the processor retry. The directory controller sends an exclusive fetch request (ERDq) to the owning nodes and changes the state to DIR_{PM}. After receiving reply, the directory controller writes it back to the local memory using BWL and changes the state to DIR_U as shown in figure 3.11. Meanwhile BLR is retried until the directory gets the block in DIR_U state. After that, the remains of operation is the same as BLR hits on DIR_U block.

Note that split locks that cross 32-byte boundaries for write back memory or 8-byte boundaries for uncacheable memory (two reads followed by two writes) could be supported since the processing node can support. However, split locks that cross remote access cache block boundaries (*e.g.* one location in a block and another in the adjacent block) are not

[6]See '5.3.4.1 [Split] Bus Lock' in [3]. The first transaction of the RMW operation may have DEFER asserted, which will retry the entire RMW operation (regardless of the response). The RMW operation is prematurely aborted and retried if the first read transaction receives an AERR assertion in the Error Phase or DEFER assertion in the Snoop Phase. After a premature abortion, the agent issuing the lock operation must ignore any data returned during Data Phase, deassert LOCK, re-arbitrate for the bus and reissue the first transaction.
See '4.4.3.1 Snoop Phase Results' in [3]. If DEFER is asserted during the Snoop Phase of a locked operation, the locked operation is prematurely aborted. During the first transaction of a locked operation, if HITM and DEFER are active together, the transaction completes with cache line writeback and implicit writeback response, but the request agent must begin a new locked operation starting from a new Arbitration Phase. The assertion of DEFER during the second or subsequent transaction of a locked operation is a protocol violation. If DEFER is asserted and HITM is not asserted, a Retry Response is driven in the Response Phase to force a retry of the entire locked operation.

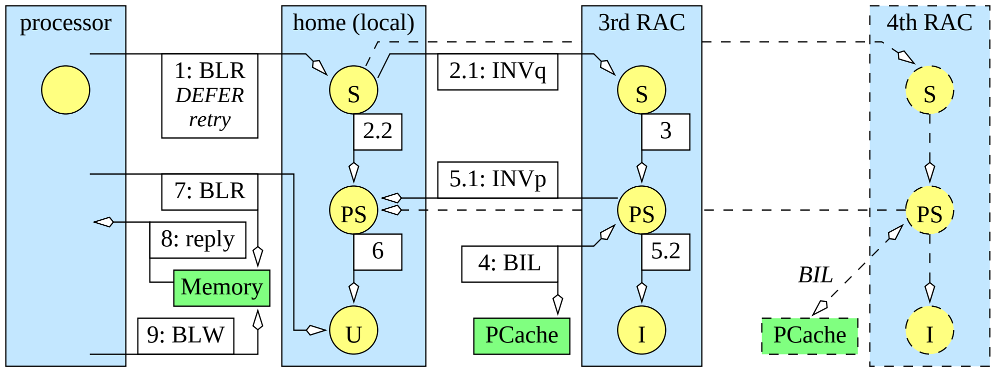


Figure 3.11: BLR hits on DIR_S

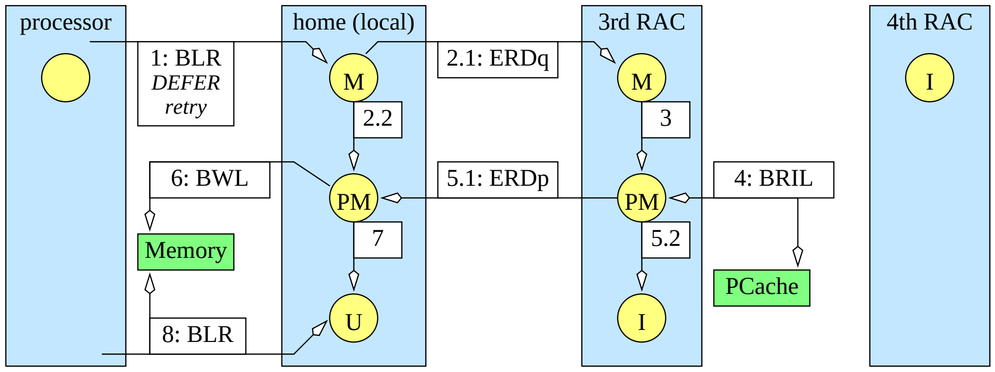


Figure 3.12: BLR hits on DIR_M

supported since the assertion of `DEFER` during the second or subsequent transaction of a locked operation is a protocol violations (see '4.4.3.1 Snoop Phase Results' in [3]). As matter of fact, split locks crossing remote access cache block boundaries could be supported if these are local accesses. However, the RACE protocol does not guarantee split bus locks.

3.8.2 Remote memory operations

In this subsection, the cases in which requesting node is not home node are explained. Therefore, all accesses dealt with in here are resolved by remote access cache.

3.8.2.1 Coherent read cases (BRL)

BRL remote request to RAC_S block reads data from the remote access cache and untouches its state. At snoop phase the RAC must drive `HIT` signal in order for the processor cache to change its state not to PC_E but to PC_S.

BRL remote request to RAC_M block reads data from the remote access cache when there is not `HITM`. Otherwise (`HITM` is asserted), the remote access cache lets processor cache provide the data and the processor also performs implicit writeback. [7] The remote access cache state does not changes.

BRL remote request missing at remote access cache makes the remote access cache send a fetch request (CRDq) to home node. At the same time, the remote access cache makes the processor wait for deferred reply by asserting `DEFER`. If the directory at the home node maintains the data in DIR_U or DIR_S state, the home node reads the data from its memory using BRL and sends it back to the requesting node and changes the state to DIR_S as shown in figure 3.13. If the home node maintains the data in DIR_M state and the home node

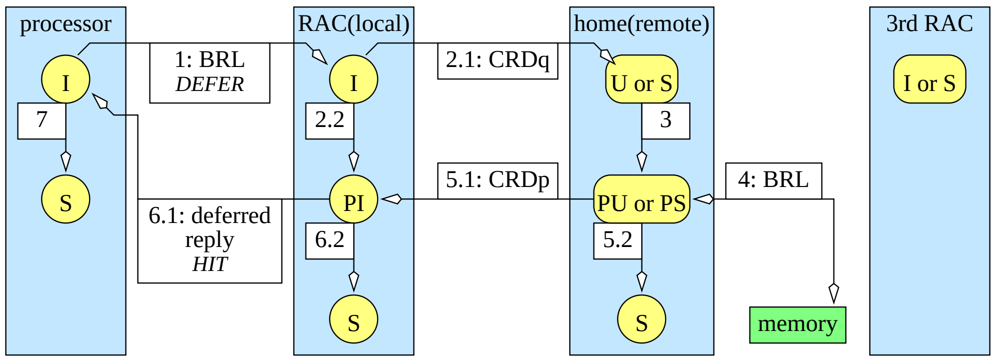


Figure 3.13: BRL misses on DIR_U or DIR_S at the home directory

forwards the request to owning node. On receiving a fetch request (CRDq), the owning node changes its state to RAC_{PM} and performs BRL to get up-to-date data. On performing BRL, if there is no `HITM` asserted by processor cache, the BRL must be terminated by the remote access cache performing a dummy response phase. Thereafter, the owning node sends the data to the home and requesting nodes and changes its state to RAC_S as shown in figure 3.14. When the reply (CRDp) arrives at the home, the home performs writeback and then changes its state to DIR_S. At the same time, when the reply (CRDp) arrives at the requesting node, the node performs deferred reply to provide the data and change its state to RAC_S. `HIT` signal must be asserted by the remote access cache to inform the data is shared one.

[7] The RAC must accept the implicit writeback in order not to lose the latest data: *e.g.* the processor performing BRL can eventually replace the line without performing writeback, since state of the lines in the processing node will be PC_S.

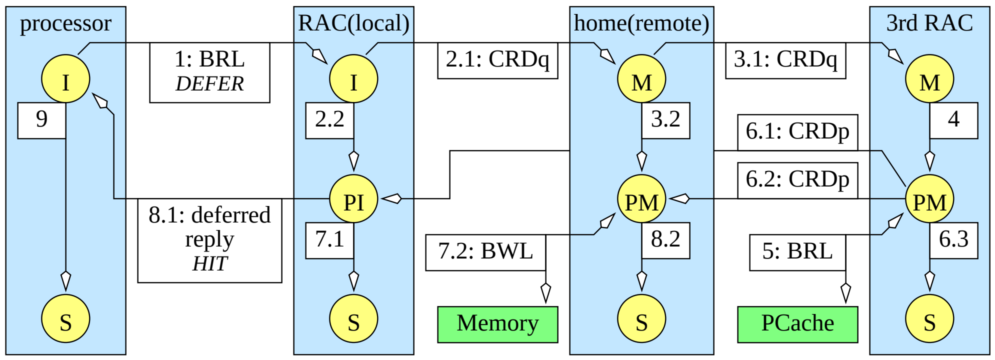


Figure 3.14: BRL misses on DIR_M at the home-directory

3.8.2.2 Exclusive read cases (BRIL)

BRIL remote request to RAC_S block makes the remote access cache send an invalidate request to the home node and change its state to RAC_{PS}. At the same time, the remote access cache makes the processor wait for deferred reply by asserting `DEFER`. On receiving the invalidate request (INVq), the directory at the home node issues invalidate requests (INVq's) to sharing nodes if any and changes its state to DIR_{PS}. The directory sends an invalidate reply (INVp) to the requesting node just after all invalidate requests have been sent. When the remote access cache at the requesting node receives the invalidate reply (INVp), it issues deferred reply on the processor bus and changes its state to RAC_M. In the meantime, the directory at the home node changes its state to DIR_M after receiving all invalidate replies as shown in figure 3.15.

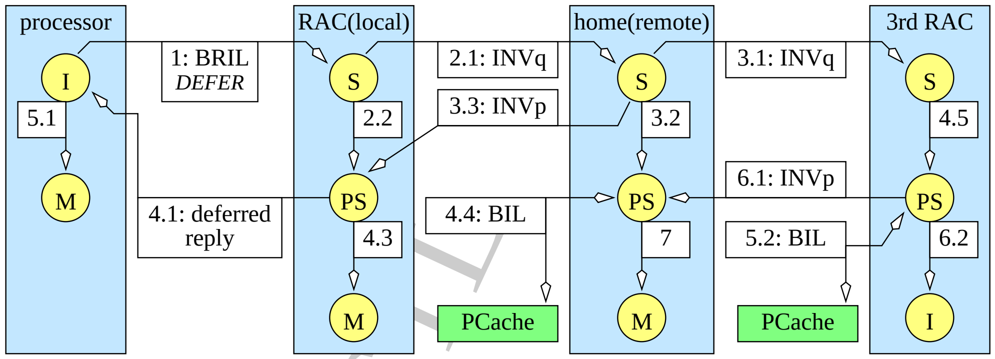


Figure 3.15: BRIL hits on RAC_S

BRIL remote request to RAC_M block reads data from the remote access cache when there is not `HITM`. Otherwise (`HITM` asserted), the remote access cache lets processor cache provide the data and the processor also performs implicit writeback. The remote access cache does not change its state.

BRIL remote request missing at remote access cache makes the remote access cache send an exclusive fetch request (ERD) to home node and change its state to RAC_{PI}. At the same time, the remote access cache makes the processor wait for deferred reply by asserting `DEFER`.

If the directory at the home node maintains the data in DIR_U state, the home node reads the data from memory using BRIL and sends it back to the requesting node and then changes the state to DIR_M as shown in figure 3.16. If the directory at the home node maintains the data

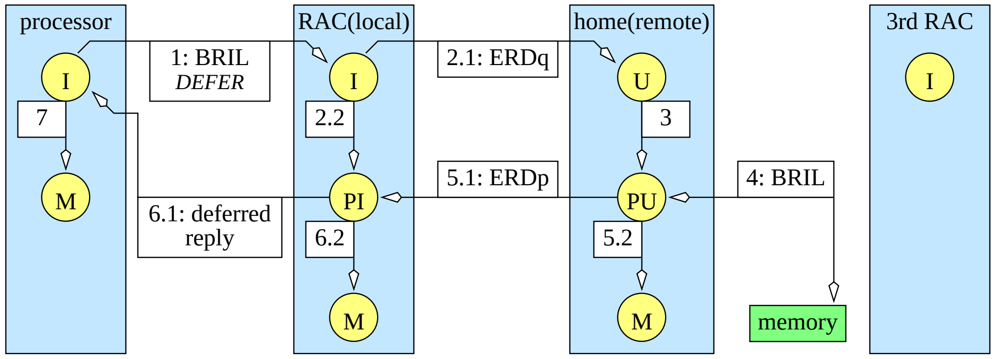


Figure 3.16: BRIL misses on DIR_U at the home directory

in DIR_S, the directory sends invalidate requests (INVq's) to sharing nodes. At the same time the directory retrieves the data by performing BRIL on the processor bus and then sends the data to the requesting node. On receiving the data, the remote access cache at the requesting node issues deferred reply and changes its state to RAC_M. In parallel with it, the directory at the home node changes its state to DIR_M after receiving all invalidate replies as shown in figure 3.17. If the directory at the home node maintains the data in DIR_M, the directory

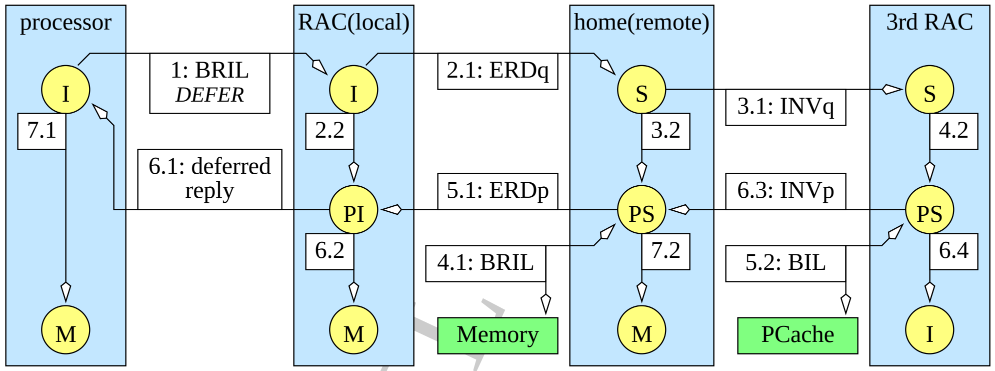


Figure 3.17: BRIL misses on DIR_S at the home directory

forwards the request to the owning node and changes its state to DIR_{PM}. On receiving an exclusive fetch request (ERDq), the remote access cache at the owning node changes its state to RAC_{PM} and performs BRIL to get up-to-date data. Thereafter, the remote access cache sends the data to the requesting node and informs the home node of the completion. and then changes its state to RAC_I. When the home node receives the reply, it changes its state to DIR_M without writeback since the up-to-date data resides in the requesting node. At the same time, when the reply arrives at the requesting node, the node performs deferred reply to provide the data and changes its state to RAC_M as shown in figure 3.18.

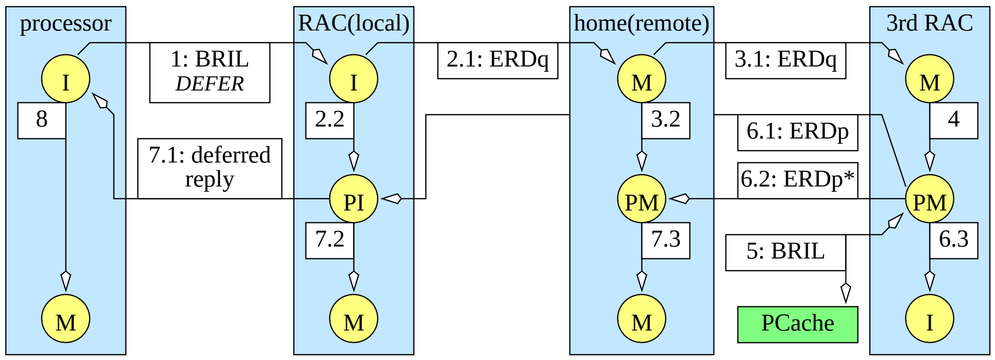


Figure 3.18: BRIL misses on DIR_M at the home directory: `6.2:ERDp*` dose not carry data.

3.8.2.3 Invalidate cases (BIL)

BIL remote request on RAC_M state at remote access cache does not affect remote access cache state since the remote access cache already has an exclusive copy. BIL remote request on RAC_I state must not occur since the processor cache in the processing node does not have copy.

BIL remote request to RAC_S block makes the remote access cache send an invalidate request to the home node and change its state to RAC_{PS}. At the same time, the remote access cache makes the processor wait for deferred reply by asserting `DEFER`. On receiving the invalidate request (INVq), the directory at the home node issues invalidate requests (INVq's) to sharing nodes if any and changes its state to DIR_{PS}. The directory sends an invalidate reply (INVp) to the requesting node just after all invalidate requests have been sent. When the remote access cache at the requesting node receives the invalidate reply (INVp), it issues deferred reply on the processor bus and changes its state to RAC_M. In the meantime, the directory at the home node changes its state to DIR_M after receiving all invalidate replies as shown in figure 3.19.

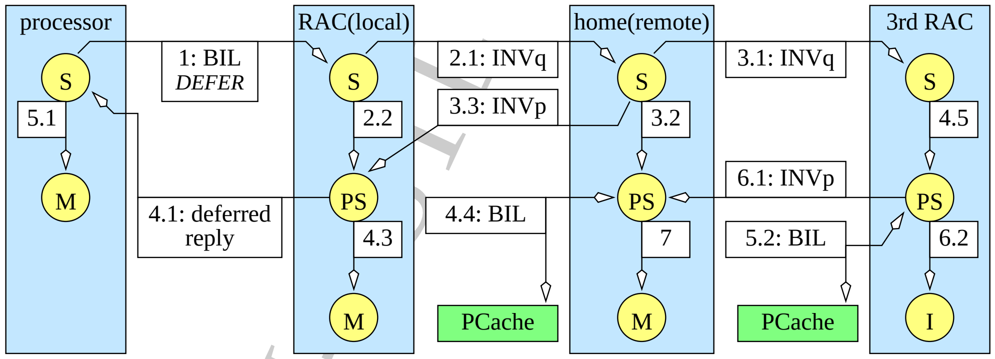


Figure 3.19: BIL hits on RAC_S

For the case of a race between BIL's from different processing nodes, only one of the BIL's will win and the winning BIL's deferred reply will not need a data transfer. However, all losing

BIL's deferred replies will need the supply the new modified data to the processor. To do this, an unexpected data reply mechanism is used. More details can be found in sections 2.4.4.5 and 3.8.1.3. Figure 3.20 shows the case of INVq hitting on DIR_{PS} as the result of BIL hitting on RAC_S. The remote access cache eventually sends ERDq instead of INVq after receiving negative acknowledgement. On receiving the data as the result of BIL, the remote access cache invalidates other processor cache lines contained at the remote access cache block before let the processor go, since the received data is most up-to-date. By doing this, no processor in the processing node can use stale data.

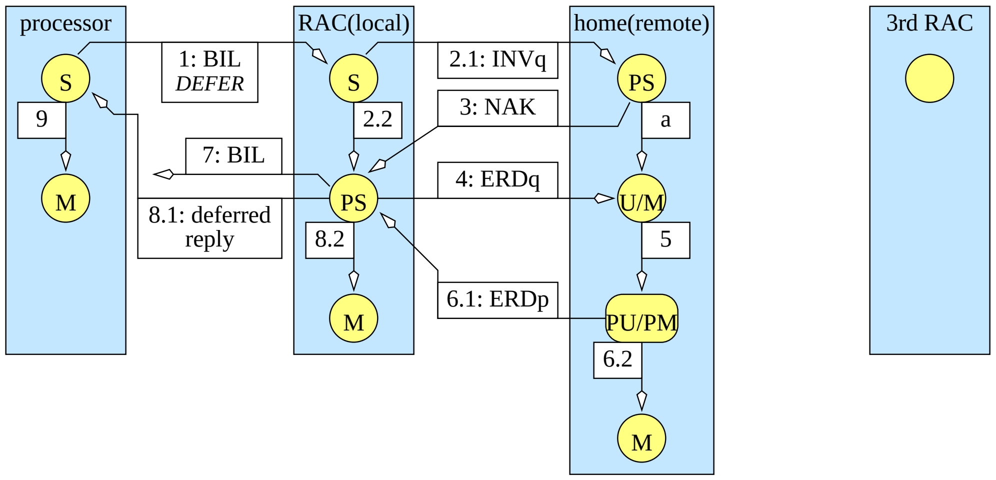


Figure 3.20: INVq hits on DIR_{PS}

3.8.2.4 Writeback cases (BWL)

BWL remote request to RAC_I or RAC_S does not occur, since the processor cache is not allowed to have PC_M. [8] BWL remote request to RAC_M just updates its remote access cache and does not propagate it to the home node as shown in figure 3.21. Note that the remote access cache must issue an BIL on the processor bus to invalidate the line inside the processor, if making sure of evicting the line[9]. In RACE implementation, an extra BIL due to BWL may not necessary, since state of the block remains in RAC_M and there is no way to indicate existence of the line inside.

In addition to these, there are two cases of writeback in the context of the remote access cache: one is explicit writeback caused by replacement of modified block and the other is implicit writeback due to network fetch request hitting on DIR_M block. The former case is shown in figure 3.22. The latter case can be divided further into the follow. If the network

[8]When `REQa[1]` is deasserted, no agent may assert `DEFER` to retry the transaction. A writeback caching agent must deassert `REQa[1]` when writing back a modified cache line to memory. If memory write transaction with `REQa[1]` asserted hits a valid line in a snooping cache, a cache coherency violation has occurred. See '5.2.1.2 Memory Write Transactions' in [3].

[9]According to Intel confidential document, when a writeback line transaction is issued (due to cache eviction) by a processor, it is not correct to assume that the evicted line is invalidated in all of the internal caches of that processors. Therefore, to guarantee that a processor does not have a cache line in a valid state after a writeback of that line, the RAC (a kind of third-party agent) must issue an extra Bus Invalidate Line transaction to invalidate the line inside the processor, upon receiving a writeback transaction.

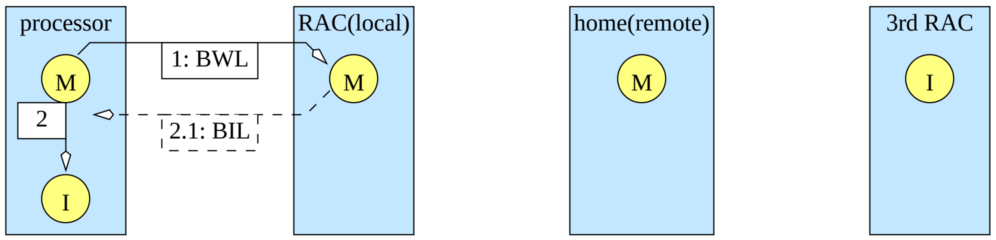


Figure 3.21: Writeback BWL

fetch request is CRD, the contents must be written back to the home node. If ERD, it depends on the initiator. If the exclusive fetch request was initiated by the home node, then the contents must be written back to the home node. Otherwise (the ERD was forwarded by the home node), the contents are forwarded to the initiator and the home node is informed of its completion. Therefore, actual writeback to the home memory does not occur.

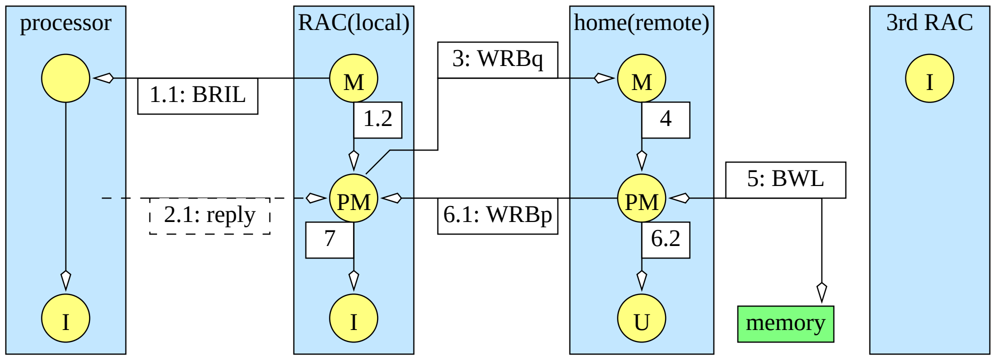


Figure 3.22: RAC writeback due to replacement

Write-back requests are generated when new remote access cache blocks are needed to make rooms for remote access cache misses. To do writeback, processor caches in requesting node must be informed that the block is going to replace using BRIL.

Generally, write-back is needed for modified remote access cache block since the contents of home memory is stale as shown in figure 3.22. The remote access cache generates BRIL request at the processor bus since it is needed to get the up-to-date data residing in the processor cache. After getting the data, the remote access cache writes it back to the home.

Writeback would be needed to prevent the home node from keeping ghost-sharing link (bit map) due to replacement of non-modified data. More detailed description can be found in section 2.4.4.4. However, the protocol does not require for the remote access cache to notify the home node of replacement of shared data (replacement-hint). Therefore, invalidate request to non-resident block must be acknowledged. It must be noted that before replacing a remote access cache block in RAC_S, all shared copy inside its processing node must be invalidated as shown in figure 3.23.

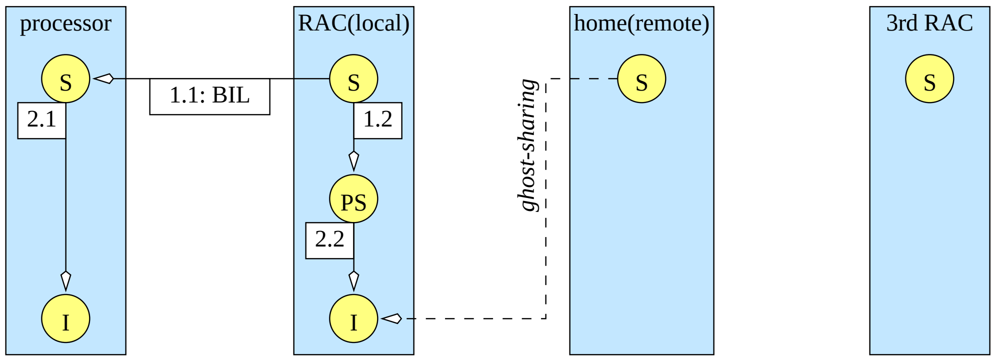


Figure 3.23: Replacement of a block in RAC_S state

3.8.2.5 Write-through cases (BWTL and BPWT)

A BWTL and BPWT requests make the remote access cache work as the same as an exclusive remote request. A BWTL and BPWT remote request on RAC_M updates the remote cache block.

A BWTL and BPWT requests on RAC_S make the remote access cache get an exclusiveness and then updates the remote cache block, as shown in figure 3.24. To do this the remote access

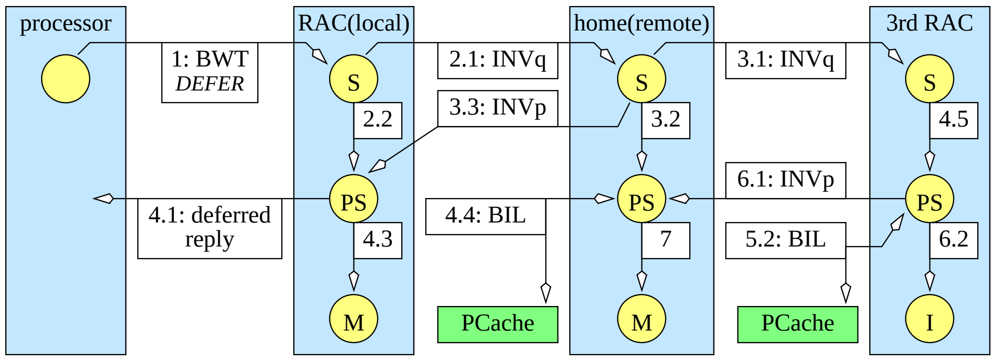


Figure 3.24: BWTL/BPWT hits on DIR_S

cache sends an invalidate request (`INV`) to the home node. On receiving the invalidate request, the home invalidates all sharing nodes and finally the remote access cache and the directory change their states to modified (M) and modified (M), respectively. [10]

A BWTL and BPWT requests on RAC_I make the remote access cache get an exclusive copy of the block and then updates it as the same as exclusive remote request does, as shown in figure 3.25.

3.8.2.6 Uncached read/write cases (BUPR and BUPW)

BUPR/BUPW remote request to RAC_S and RAC_M block does not occur, since RAC_S and RAC_M mean that the block has been accessed as a cacheable region.

[10]When writing through the cache, `REQa[1]` is asserted. This transaction is snooped and can receive an implicit writeback response. See '5.2.1.2 Memory Write Transactions' in [3].

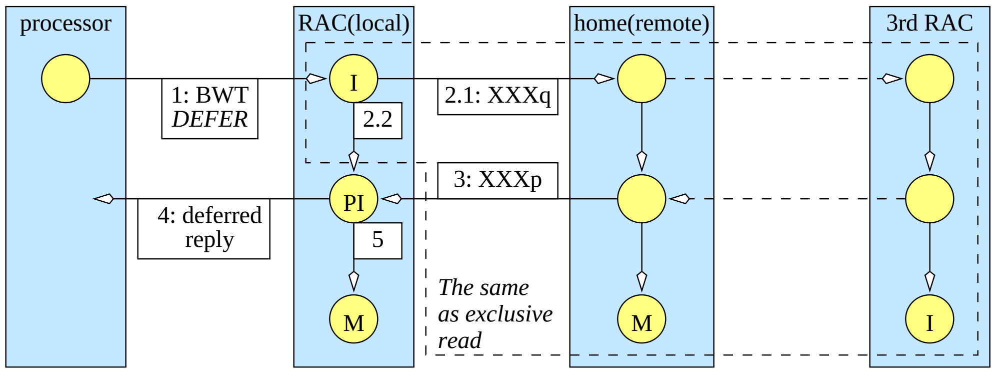


Figure 3.25: BWTL/BPWT hits on RAC_I

An BUPR remote request generates a uncached read request (URD) to home node. If the state at the home node's directory is not DIR_U, a protocol violation occurs. The directory retrieves a copy from its memory and sends it back to the requestor as shown in figure 3.26.

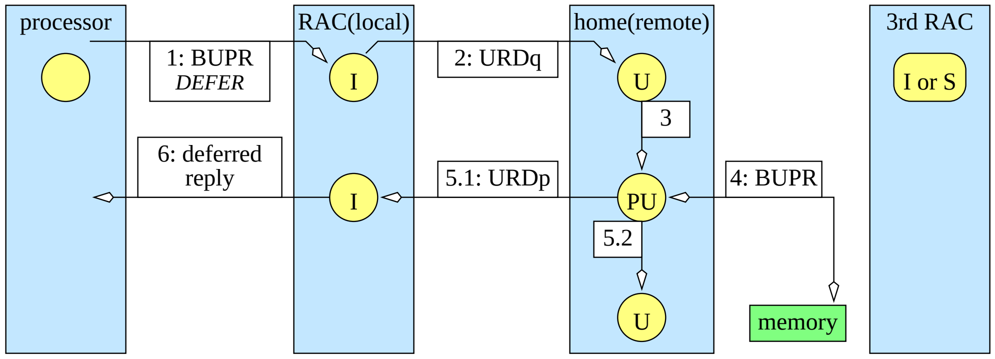


Figure 3.26: BUPR remote request to RAC_I and DIR_U

An BUPW remote request generates a uncached read request (UWR) to home node. If the state at the home node's directory is not DIR_U, a protocol violation occurs. The directory updates its memory and sends the response back to the requestor as shown in figure 3.27.

3.8.2.7 Locked read and write cases (BLR and BLW)

A pair of BLR and BLW remote requests must keep out all other references to the same address between them. To do this, BLR remote request must be performed on RAC_M or RAC_L state meaning that the line is valid and exclusive at remote access cache. BLW remote request following the BLR must hit on the remote cache with RAC_L state. Otherwise, atomicity of read-modified-write cannot be guaranteed.

BLR remote request to RAC_S block makes the processor retry[11]. (The remote access

[11]See '5.3.4.1 [Split] Bus Lock' in [3]. The first transaction of the RMW operation may have `DEFER` asserted, which will retry the entire RMW operation (regardless of the response). The RMW operation is prematurely aborted and retried if the first read transaction receives an `AERR` assertion in the Error Phase or `DEFER` assertion in the Snoop Phase. After a premature abortion, the agent issuing the lock operation must ignore any data

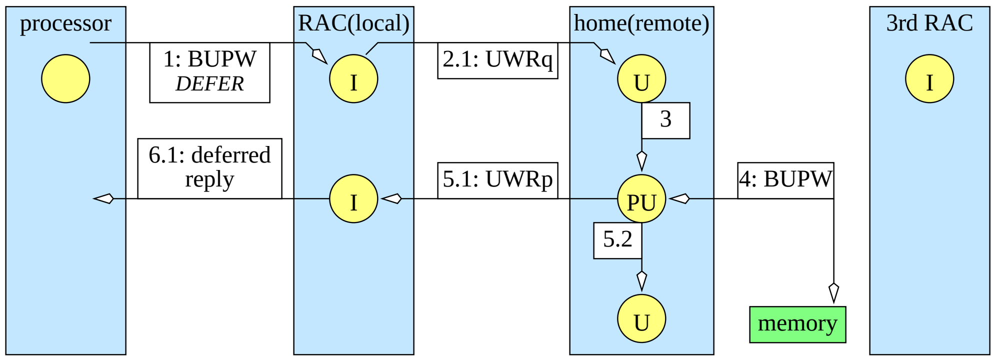


Figure 3.27: BUPW remote request to RAC_I and DIR_U

cache asserts DEFER followed by a retry response to abort the bus lock sequence.) The remote access cache sends invalidate request (INVq) to home node and changes its state to RAC_{PS}. On receiving the invalidate request, directory at the home node issues invalidate requests (INVq's) to sharing nodes if any and changes its state to DIR_{PS}. The directory sends an invalidate reply (INVp) to the requesting node just after all invalidate requests have been sent. When the remote access cache at the requesting node receives the invalidate reply (INVp), it changes its state to RAC_L. In the meantime, the directory at the home node changes its state to DIR_M after receiving all invalidate replies. Subsequently, the retried BLR hits on RAC_L block and gets data replied. The following BLW also hits on RAC_L block and changes the state to RAC_M as shown in figure 3.28.

BLR remote request to RAC_M state at remote access cache reads the data from the remote access cache which changes states to RAC_L. The following BLW also hits on RAC_L block and changes the state to RAC_M as shown in figure 3.29.

BLR remote request missing at remote access cache makes the remote access cache send an exclusive fetch request to home node and change its state to RAC_{PI}. At the same time, the remote access cache makes the processor retry. If the directory at the home node maintains the data in DIR_U state, the home node reads the data from memory using BRIL and sends it back to the requesting node and then changes the state to DIR_M as shown in figure 3.30. If the directory at the home node maintains the data in DIR_S, the directory sends invalidate request (INVq) to sharing nodes. At the same time the directory retrieves the data by performing BRIL on the processor bus and then sends the data to the requesting node. On receiving the data, the remote access cache at the requesting node changes its state to RAC_L. In parallel with it, the directory at the home node changes its state to DIR_M after receiving all invalidate replies as shown in figure 3.31. If the directory at the home node maintains the data in DIR_M, the directory forwards the request to the owning node and changes its state to DIR_{PM}. On receiving an exclusive fetch request (ERDq), the remote access cache

returned during Data Phase, deassert LOCK, re-arbitrate for the bus and reissue the first transaction. See '4.4.3.1 Snoop Phase Results' in [3]. If DEFER is asserted during the Snoop Phase of a locked operation, the locked operation is prematurely aborted. During the first transaction of a locked operation, if HITM and DEFER are active together, the transaction completes with cache line writeback and implicit writeback response, but the request agent must begin a new locked operation starting from a new Arbitration Phase. The assertion of DEFER during the second or subsequent transaction of a locked operation is a protocol violation. If DEFER is asserted and HITM is not asserted, a Retry Response is driven in the Response Phase to force a retry of the entire locked operation.

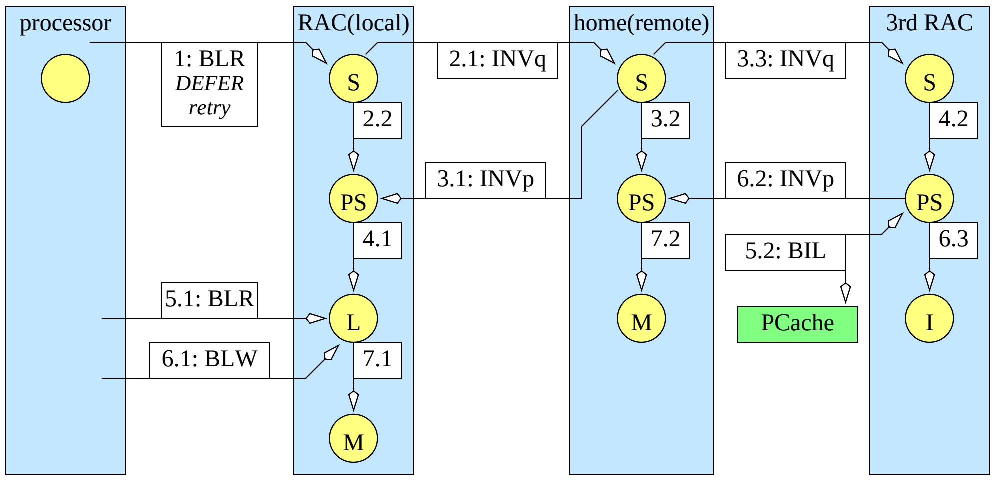


Figure 3.28: BLR hits on RAC_S

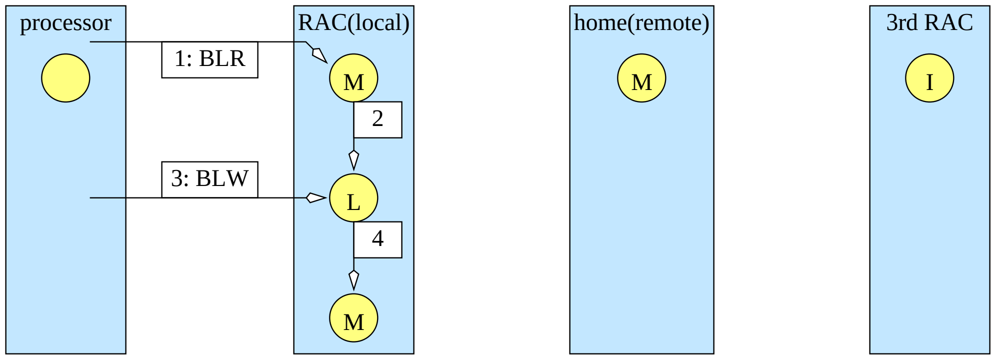


Figure 3.29: BLR hits on RAC_M

at the owning node changes its state to RAC_{PM} and performs BRIL to get up-to-date data. Thereafter, the remote access cache sends the data to the requesting node and informs the home node of the completion, and then changes its state to RAC_I. When the home node receives the reply, it changes its state to DIR_M without writeback since the up-to-date data resides in the requesting node. At the same time, when the reply arrives at the requesting node, the node changes its state to RAC_L as shown in figure 3.32. Subsequently, the retried BLR hits on RAC_L block. The following BLW also hits on RAC_L block and changes the state to RAC_M.

Note that, remote split lock that cross remote access cache block boundaries (*e.g.* one location in a line and another in the adjacent line) are not supported since the assertion of `DEFER` during the second or subsequent transaction of a locked operation is a protocol violations (see '4.4.3.1 Snoop Phase Results' in [3]).

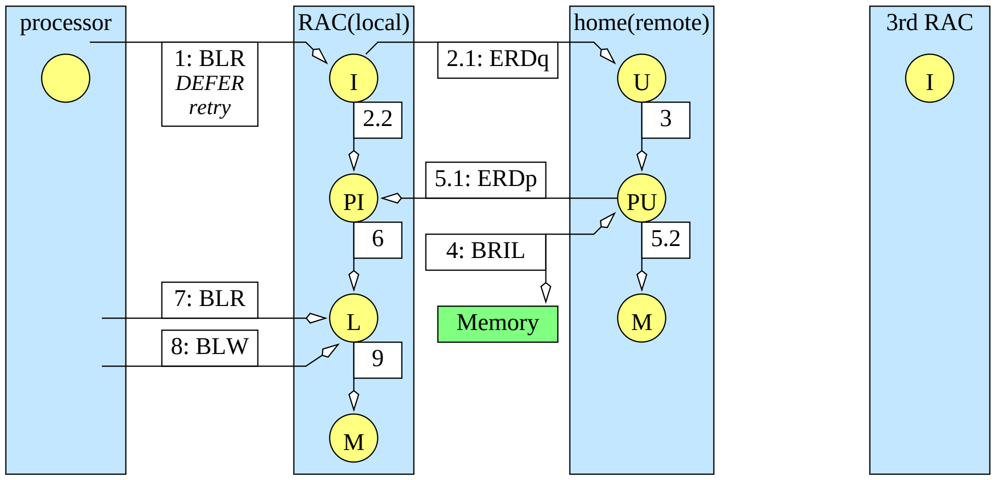


Figure 3.30: BLR misses on DIR_U at the home directory

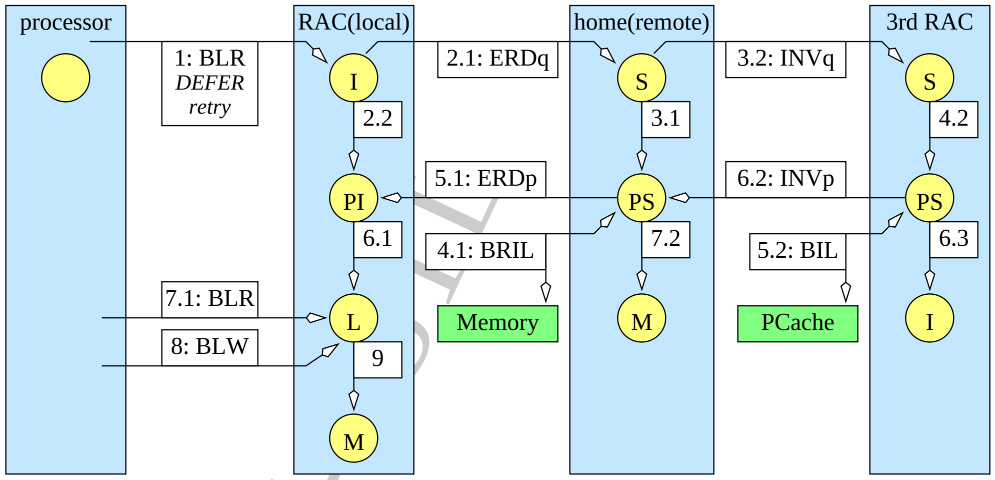


Figure 3.31: BLR misses on DIR_S at the home directory

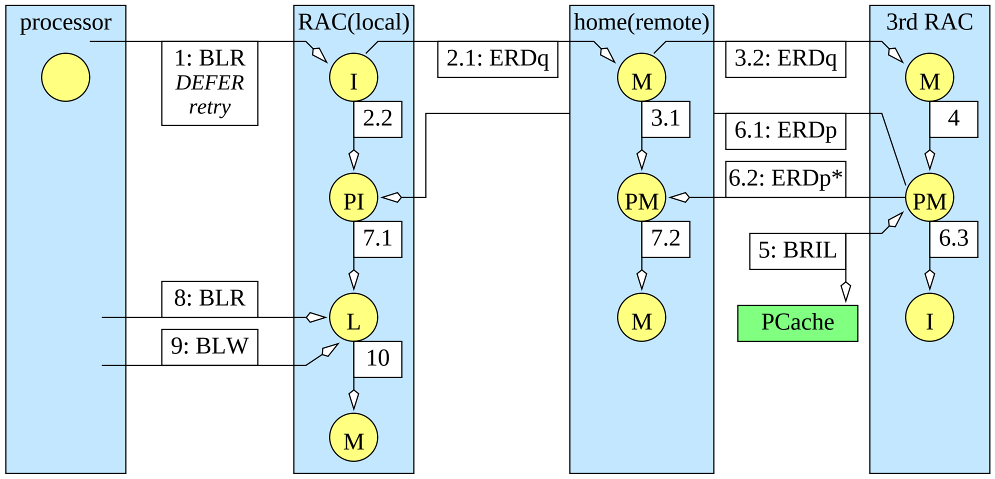


Figure 3.32: BLR misses on DIR_M at the home directory: `6.2:ERDp*` does not carry data.

References

[1] Jean-Loup Baer and Wen-Hann Wang. Architectural choices for multilevel cache hierarchies. In *Proceedings of the 1st International Conference on Parallel Processing*, pages 258–261, August 1987.

[2] Intel. *Pentium® II Processor Developer's Manual.* Intel, October 1997. Order Number: 243502-001.

[3] Intel. *Pentium® Pro Family Developer's Manual, Volume1: Specifications.* Intel, 1997. Order Number: 242690.

[4] Intel. *Pentium® Pro Family Developer's Manual, Volume3: Operating System Writer's Manual.* Intel, 1997. Order Number: 242692.

[5] Intel. *AD450NX MP Server Board Set Technical Specification.* Intel, August 1998. Order Number: 723177-001.

[6] Intel. *Intel® 450NX PCIset.* Intel, June 1998. Order Number: 243771-004.

[7] Intel. *Pentium® II Xeon™ Processor At 400 MHz.* Intel, 1998. Order Number: 243770-001.

[8] Ando Ki, Byung Kwan Park, Won Sae Sim, Kyeng Yong Kang, and Yong Ho Yoon. Highly Pipelined Bus : HiPi-Bus. In *Proceedings of the Joint Technical Conference*, pages 528–533, 1991. Hirosima Japan, (an extended version also appeared in ETRI Journal, 13(3), p.33–50, Oct. 1991).

[9] Paul Sweazey and Alan Jay Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, 1986.

Index

[A]
agent 6

[B]
block 9
bus lock mechanism 27, 53
BusCoRd . . . . . . 6, *see* intra-node network operation
BusExRd . . . . . . 6, *see* intra-node network operation
BusInv6, *see* intra-node network operation
BusIoRd . . . . . . 7, *see* intra-node network operation
BusLocRd . . . . . . 6, *see* intra-node network operation
BusLocWr . . . . . . 6, *see* intra-node network operation
BusUcRd . . . . . . 6, *see* intra-node network operation
BusUcWr . . . 6, *see* intra-node network operation, 7, *see* intra-node network operation
BusWrb . . . . . . 6, *see* intra-node network operation
BusWrt . . . . . . 6, *see* intra-node network operation

[C]
cache 2
  cache coherency protocol 3
  no-write-allocate 3
  write-allocate 3
  write-through 3
  writeback 3
cache coherency protocol . . . . . 3, *see* cache
  directory protocol 3
  snoopy protocol 3
cache state
  directory 10
  Pentium® Pro 43
  processor cache 9
  remote access cache 9
cache-coherent non-uniform-memory-access *see* shared-address architecture
cache-only-memory architecture . . . . . . *see* shared-address architecture
CC-NUMA . . . . . . *see* cache-coherent non-uniform-memory-access
clean 9
coherent agent 19
COMA . . . . . . *see* cache-only-memory architecture
CRD . 7, *see* inter-node network operation
critical-line-first 50
critical-word-first 50

[D]
`DEFER` 43
deferred transaction 36
  `DEFER` 36
DIR . . . . . . *see* directory
directory . . *see* figure 2.1, 10, *see* figure 2.2
directory protocol . . 3, *see* cache coherency protocol
directory state 10
  modified 10
  shared 10
  uncached 10

[E]
ERD . . 7, *see* inter-node network operation
exclusive 9, 43
  Pentium® Pro 43
  processor cache state 9

[F]
front side bus 34
  six phases 35
FSB . . . . . . *see* front side bus

full-map scheme .... *see* directory protocol

[G]

ghost-sharing 10, *see* replacement-hint, 17, 28, 60

[H]

HIT 43
HITM 43
home node 17

[I]

implicit writeback 43, 55, 56
in-order completion 36
in-order message delivery 17, 31
in-order queue 36
inter-node network operation 7
    CRD 7
    ERD 7
    INV 7
    IOR 8
    IOW 8
    URD 7
    UWR 8
    WRB 7
intra-node network operation 6
    BusCoRd 6
    BusExRd 6
    BusInv 6
    BusIoRd 7
    BusIoWr 7
    BusLocRd 6
    BusLocWr 6
    BusUcRd 6
    BusUcWr 6
    BusWrb 6
    BusWrt 6
INV .. 7, *see* inter-node network operation
invalid 9, 10, 43
    Pentium® Pro 43
    processor cache state 9
    remote access cache state 10
IOQ .... *see* in-order queue
IOR .. 8, *see* inter-node network operation
IOW .. 8, *see* inter-node network operation

[L]

line 9
livelock 29
local access .... 17, *see* figure 2.2(a), 19
    BIL 51
    BLR 53
    BLW 53
    BPWT 52
    BRIL 50
    BRL 49
    BUPR 52
    BUPW 52
    BWL 52
    BWTL 52
    coherent local request 19
    coherent read 49
    exclusive local request 19
    exclusive read 50
    invalidate 51
    invalidate local request 20
    local locked access 27
    locked read 53
    locked write 53
    uncached read 52
    uncached read local request 21
    uncached write 52
    uncached write local request 21
    write-through 52
    write-through local request 21
    writeback 52
    writeback local request 21
local locked access 27
local memory 17
local node 17
local request .... *see* local access
lock 27
locked 10
    remote access cache state 10

[M]

memory and I/O controller 34
memory type 37
    Pentium® Pro 37
    uncacheable 38
    write back 38
    write combining 38
    write protected 38
    write through 38
MESI 9, 43

PDLSystem 9
Pentium® Pro 43
message-passing . . *see* programming model
MIOC . . . . . *see* memory and I/O controller
modified 9, 10, 43
directory state 10
Pentium® Pro 43
processor cache state 9
remote access cache state 10
multi-level inclusion 9

[**N**]
NAK 8
NAMA . . *see* network-abstraction-memory architecture
network interface . . . . . 5, *see* figure 2.1
network operation
inter-node 7
intra-node 6
network-abstraction-memory architecture*see* shared-address architecture
NIF . . . . . *see* network interface
no-write-allocate . . . . . *see* cache
non-uniform-memory-access . . . . . *see* shared-address architecture
NUMA . . . *see* non-uniform-memory-access

[**O**]
operation 6
owning node 17

[**P**]
part-line transaction . 42, *see* transfer type
PDLSystem . *see* physically-distributed but logically-shared memory system
pending 10, 11
directory 11
remote access cache 10
physically-distributed but logically-shared memory system . . . . . 1, *see* shared-address architecture, 5
point-to-point order 17
seein-order message delivery 17
processor cache state 9
exclusive 9
invalid 9
modified 9
Pentium® Pro 43
shared 9
programming model
message-passing 1
shared-memory 1

[**R**]
RAC . . . . . *see* remote access cache
RACE . *see* remote access cache coherency enforcement
read-modify-write 27
remote access . . . . . 17, *see* figure 2.2(b)
BIL 58
BLR 62
BLW 62
BPWT 61
BRIL 56
BRL 55
BUPR 61
BUPW 61
BWL 59
BWTL 61
coherent read 55
exclusive read 56
invalidate 58
invalidate remote request 23
locked read 62
locked write 62
remote locked access 27
uncached read 61
uncached read remote request 25
uncached write 61
uncached write remote request 25
write-through 61
write-through remote request 24
writeback 59
writeback remote request 24
remote access cache . . *see* figure 2.1, 9, *see* figure 2.2
remote access cache coherency enforcement 17, 49
remote access cache state 9
invalid 10
locked 10
modified 10
shared 10
remote accesses 22
coherent remote request 22

exclusive remote request 22
remote locked access 27
remote memory 17
remote node 17
remote request . . . . *see* remote access
replacement 28
remote access cache block 28
replacement-hint 10, *see* ghost-sharing, 28, 60
reply-forwarding 17
CRD 22, 55
ERD 23, 57
error 22, 23
retry 22, 23
requesting node 17
requestor 6
retry 53
RMW . . . . *see* read-modify-write
RPL 8

[**S**]
shared 9, 10, 43
directory state 10
Pentium® Pro 43
processor cache state 9
remote access cache state 10
shared-address architecture
cache-coherent non-uniform-memory-access 2
cache-only-memory architecture 2
network-abstraction-memory architecture 2
non-uniform-memory-access 2
shared-virtual-memory architecture . 2
uniform-memory-access 2
shared-memory . . . *see* programming model
shared-virtual-memory architecture . . . . *see* shared-address architecture
SHV . . . . *see* standard high volume
six phases 35
snoopy protocol . . . . 3, *see* cache coherency protocol
space inclusion . . . . *see* space multi-level inclusion
space multi-level inclusion 9
split bus lock 54
split-lock 48
standard high volume 34
state 9, 10
directory 10
processor cache 9
remote access cache 9
SVMA . . . . *see* shared-virtual-memory architecture
synchronization 27

[**T**]
third-party agent 34
TPA . . . . *see* third-party agent
TPCTL . . . . *see* table 3.1
`TPCTL` 37
transaction 6
transfer type 39
BIL 40
BIOR 41
BIOW 41
BLR 41
BLW 41
BPR 40
BPW 40
BPWT 40
BRIL 40
BRL 40
BRP 40
BUPR 41
BUPW 41
BURL 41
BURP 41
BUWL 41
BUWP 41
BWL 40
BWP 40
BWTL 40
BWTP 40
Pentium® Pro 39
two steps approach 44

[**U**]
UC . . . . *see* uncacheable
UMA . . . . *see* uniform-memory-access
uncacheable . . . . *see* memory type
uncached 10
directory state 10

unexpected data reply mechanism . 30, 51, 59
uniform-memory-access *see* shared-address architecture
URD . 7, *see* inter-node network operation
UWR . 8, *see* inter-node network operation

[**V**]
valid 9

[**W**]
WB . . . *see* write back, *see* write protected
WC . . . . *see* write combining
WRB . 7, *see* inter-node network operation
write back . . . . *see* memory type
write combining . . . . *see* memory type
write protected . . . . *see* memory type
write through . . . . *see* memory type
write-allocate . . . . *see* cache
write-through . . . . *see* cache, writeback, *see* memory type
writeback . . . . *see* cache, write-through, *see* memory type
WT . . . . *see* write through