# Performance Comparison of Cache Coherence Protocol on Multi-Core Architecture

**Anoop Tiwari**
(Roll No: 212CS1100)

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela, Odisha, 769008, India

# Performance Comparison of Cache Coherence Protocol on Multi-Core Architecture

Dissertation submitted in

*May 2014*

*to the department of*

***Computer Science and Engineering***

*of*

***National Institute of Technology Rourkela***

*in partial fulfillment of the requirements*

*for the degree of*

***Master of Technology***

*by*

***Anoop Tiwari***

*(Roll 212CS1100)*

*under the supervision of*

***Dr. Ashok Kumar Turuk***

**Department of Computer Science and Engineering**
**National Institute of Technology Rourkela**
**Rourkela – 769 008, India**

Computer Science and Engineering
**National Institute of Technology Rourkela**
Rourkela-769 008, India. `www.nitrkl.ac.in`

**Dr. Ashok Kumar Turuk**

May , 2014

## Certificate

This is to certify that the work in the thesis entitled *Performance Comparison of Cache Coherence Protocol on Multi-Core Architecture* by *Anoop Tiwari*, bearing roll number 212CS1100, is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of *Master of Technology* in *Computer Science and Engineering*. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

***Ashok Kumar Turuk***

## Acknowledgement

This dissertation, though an individual work, has benefited in various ways from several people. Whilst it would be simple to name them all, it would not be easy to thank them enough.

First of all, I would like to express my deep sense of respect and gratitude towards my supervisor *Dr. Ashok Kumar Turuk*, who has been the guiding force behind this work. I want to thank him for introducing me to the field of Multi-core Technology and giving me the opportunity to work under him. His undivided faith in this topic and ability to bring out the best of analytical and practical skills in people has been invaluable in tough periods. Without his invaluable advice and assistance it would not have been possible for me to complete this thesis. I am greatly indebted to him for his constant encouragement and invaluable advice in every aspect of my academic life. I consider it my good fortune to have got an opportunity to work with such a wonderful person.

It is indeed a privilege to be associated with people like *Prof. S. K. Rath*, *Prof. S.K.Jena*, *Prof. D. P. Mohapatra*, *Prof. A. K. Turuk*, *Prof. S.Chinara*, *Prof. Pankaj Sa* and *Prof. B. D. Sahoo*. They have made available their support in a number of ways.

I wish to thank all faculty members and secretarial staff of the CSE Department for their sympathetic cooperation.

During my studies at N.I.T. Rourkela, I made many friends. I would like to thank them all, for all the great moments I had with them.

When I look back at my accomplishments in life, I can see a clear trace of my family's concerns and devotion everywhere. My dearest father, whom I owe everything I have achieved and whatever I have become; my beloved mother, for always blessing me and inspiring me even at the toughest moments of my life; and my brother and sister; who were always my silent support during all the hardships of this endeavor and beyond.

*Anoop Tiwari*

## Declaration

I, Anoop Tiwari (Roll No. 212CS1100) understand that plagiarism is defined as any one or the combination of the following

1. Uncredited verbatim copying of individual sentences, paragraphs or illustrations (such as graphs, diagrams, etc.) from any source, published or unpublished, including the internet.
2. Uncredited improper paraphrasing of pages or paragraphs (changing a few words or phrases, or rearranging the original sentence order).
3. Credited verbatim copying of a major portion of a paper (or thesis chapter) without clear delineation of who did or wrote what.(Source: IEEE, the Institute, Dec. 2004)

I have made sure that all the ideas, expressions, graphs, diagrams, etc., that are not a result of my work, are properly credited. Long phrases or sentences that had to be used verbatim from published literature have been clearly identified using quotation marks.
I affirm that no portion of my work can be considered as plagiarism and I take full responsibility if such a complaint occurs. I understand fully well that the guide of the thesis may not be in a position to check for the possibility of such incidences of plagiarism in this body of work.

**Date:**

**Anoop Tiwari**
Master of Technology
Computer Science and Engineering
NIT Rourkela

# Contents

## List of Figures

## List of Tables

# Abstract

Number of cores in multi-core processors is steadily increased to make it faster and more reliable. Increasing the number of cores comes with a numerous issues that need to be addressed. In this dissertation we looked at the cache coherence issue, its importance and solution. Cache coherence is important as two or more cores sharing the same data must maintain the recent updated value to avoid reading of stale value. We have made an extensive study of existing cache coherence methods, such as Snoopy coherence technique and Directory coherence technique. Snoopy coherence technique is studied with the help of MOESI coherence protocol and Directory coherence technique is observed with the help of MI, MESI_TWO_LEVEL, MESI_THREE_LEVEL, MOESI, and MOESI TOKEN coherence protocol. We have used GEM5 simulator and Splash-2 benchmark to compare their performance. For simulation a precompiled program called MemTest, Ruby_random_tester, and Splash-2 suite is used. It is observed that the performance is improved as we move from MI, MESI_TWO_LEVEL, MESI_THREE_LEVEL, MOESI, and MOESI TOKEN in Directory coherence technique and for Snoopy coherence we observed the performance through varying parameters like, cache size, block size and associativity. It is also observed that that adding L3 level cache the performance of MESI_Three_Level is improved over MESI_Two_Level.

**Keywords:** Multi-core processors, Cache Coherence, Snoopy Coherence technique, Directory Coherence technique.

# Chapter 1

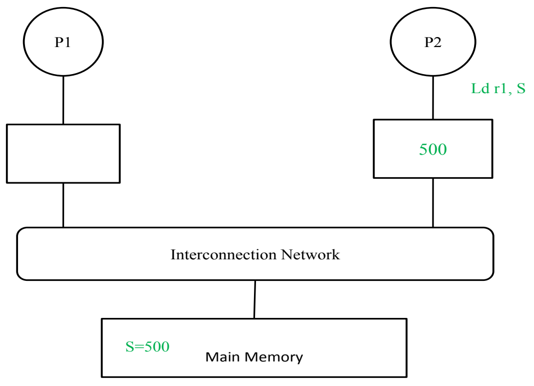
## 1 Introduction and Motivation

The performance of computer, web servers are very crucial today as they, becoming part of lives. Earlier the concept was increasing processer's frequency means a faster computer. But after saturation in increasing processor's frequency, the designers need to focus on other parameters like increasing number of cores to achieve high performance [22]. Many of today's computers are using multiprocessors with shared memory. Shared memory multiprocessors need to be coherent and consistent with data [21]. Cache coherence is critical issue in multi-core system Cache coherence problem is defined as when two different cores can have two different values for the same location. Coherence problem exist because we have both a global state, defined primarily by the main memory, and a local state, defined by the individual caches, which are private to each core [15]. Cache coherence protocols are studied and applied, to deal with the problem of coherence inside shared-memory multiprocessors. Coherence protocols coordinate distributed caches, so as to provide consistent view of memory to processors [19].

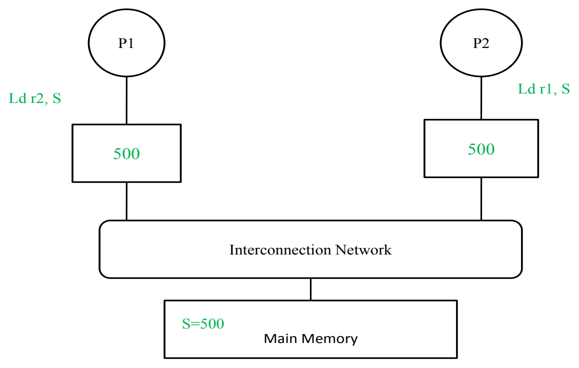
This chapter describes conventional coherence protocols (section1.1), requirements for better coherence protocol (section 1.2), contribution of this thesis (section1.3) and then chapter ends with structure of thesis (section 1.4).

## 1.1 Cache Coherence Protocol:

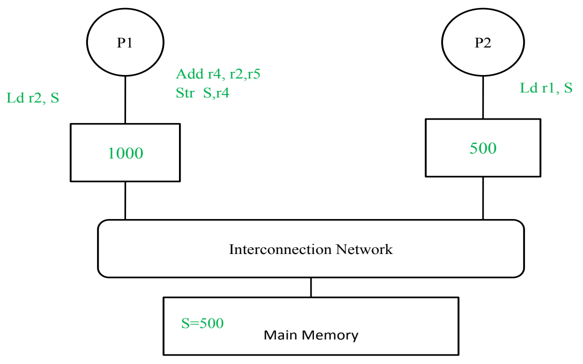
Cache coherence protocol is solution for maintaining consistency between the shared cache block. Figure 1(a), 1(b), 1(c) and 1(d) shows briefly the coherence problem.



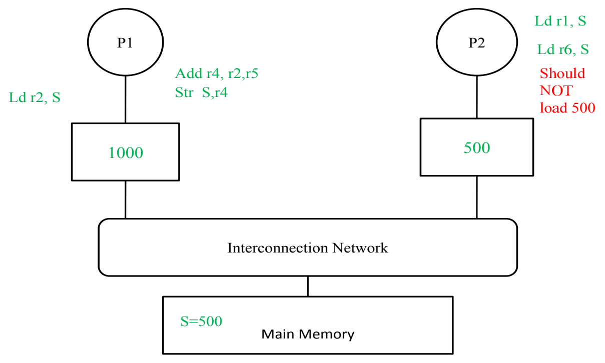
(a) Variable S read by processor P2



(b) Variable S read by processor P1 for update



(c) Value of S updated by P1



(d) P2 should read updated value

Figure 1: Coherence Problem

Now, either we can maintain cache coherence with software or hardware approach. When we talk about software approach then the programmer has to ensure cache coherence. Other software approach use cache flush instruction in ISA (instruction set architecture) to flush or invalidate cache blocks shared among processors. But these techniques make programmer's work harder So we move towards hardware solution, where to maintain cache coherence, updates made by one processor should be communicated to others and there should be consistent view of memory by all the processors. The basic idea is that the processors should broadcast its operation to all other processors and other processors should change state of shared block accordingly. Earlier approach use two type of protocol: **1: Update** and **2: Invalidate**. In update protocol, when write operation is performed, a broadcast message is sent to other sharers of block, so that they update their cache block. In invalidation protocol write operation performed by one processor, broadcast invalidation message to sharers and sharers clear their shared block accordingly. There are drawbacks with each above approaches. If data block is present in only one cache then after each update, broadcasting of "update" message is useless in Update method. Similarly if core itself do update and read operation then broadcasting invalidate message is useless in "Invalidate" method. There was requirement for proper update protocol. Snoopy based technique developed by Goodman in 1983 and Directory based technique developed by Censier and Feautrier in IEEE Toc 1978, are two most studied cache coherence protocols used today. In snoopy based method, a bus based communication is used to broadcast messages to other connected processors. Through bus processors observe requests/response send by other processors. For example: if P1 perform write operation on one block then the invalidate message is sent to other processors and processor sharing same block, clear its block. Figure 2, describes snoopy protocol method.

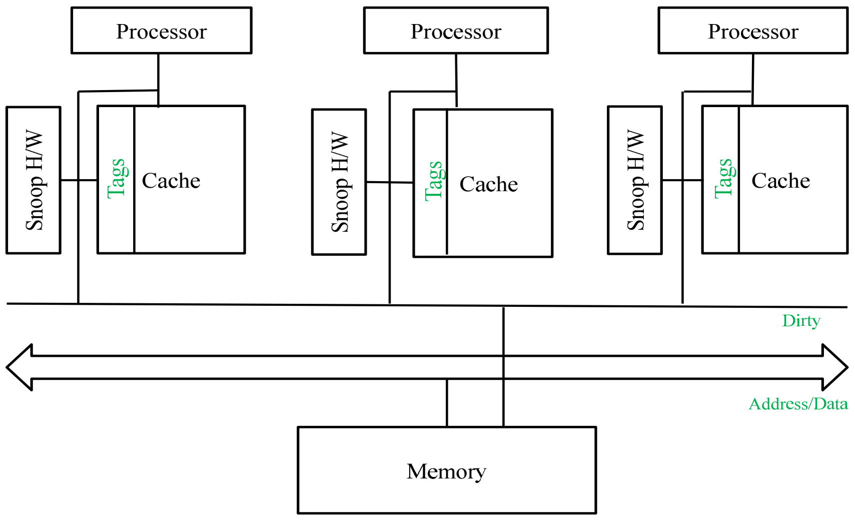


Figure 2: Snoopy Coherence Method

In directory based protocol, directory tracks state for each block. Request generated by processors for a block, firstly direct the request to directory containing block. Then the directory either forwards the request or serves the request depending on state of block. Directory communication is a kind of point-to-point communication [1]. Figure 3, describe directory method [16].

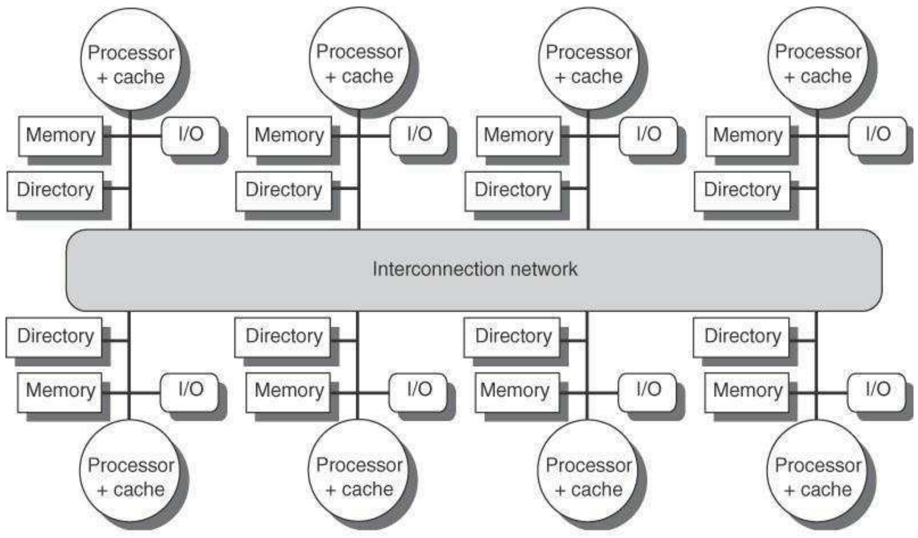


Figure 3: Directory Coherence Method

## 1.2 Requirements for Proper coherence protocol:

For any good coherence protocol there should is some important features need to be followed. Like, there should be low latency for cache to cache transfer, Bus based communication need to be avoided and a protocol should be bandwidth efficient. **First**, cache-to-cache transfer is very important as we prefer transfer of required data through cache-to-cache instead of transferring through main memory. Snoopy based method broadcast messages on bus, hence support low latency cache-to-cache transfer. In contrast, directory based protocol direct request to destination cache and wait for acknowledgement which takes extra clock cycle. **Second**, avoidance of dependency on bus like architecture as it restricts integration of more cores on the system. Directory based protocol has benefit of integrating large number of cores as here communication is point-to-point. **Third**, bandwidth efficiency can only be mitigated up to some level. Bandwidth need to be reduced to avoid interconnect contention, as it affects system performance. Below triangle diagram describe desirable features [20].

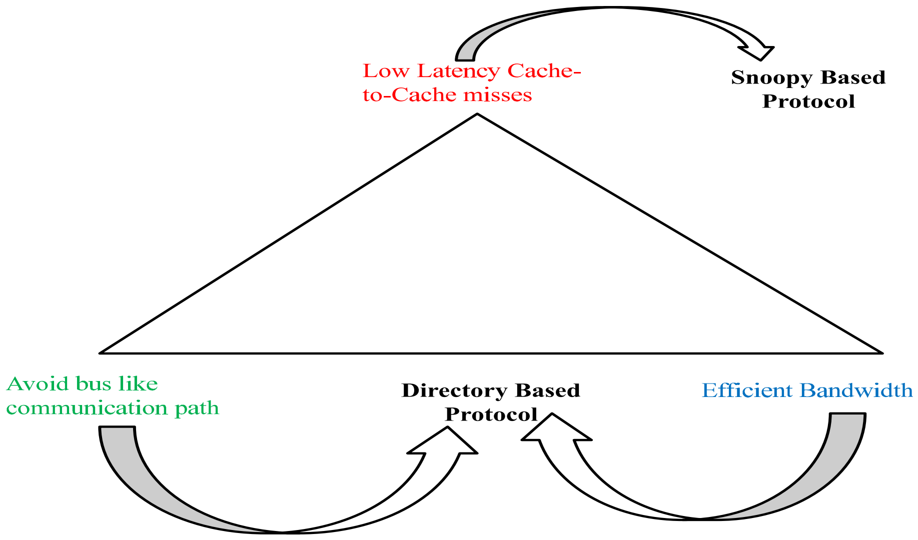

Low Latency Cache-to-Cache misses
Snoopy Based Protocol
Avoid bus like communication path
Directory Based Protocol
Efficient Bandwidth

Figure 4: Coherence Requirements

Triangle vertices represent different desirable features. Each feature is linked to its corresponding protocol.

## 1.3 Contribution of thesis work:

This section highlights our work done in the field of cache coherence.

- **Studied coherence protocol and its practical implementation:** In this thesis I have presented cache coherence techniques, recent coherence protocol use in Intel Nehalem architecture and how can we test our own designed protocol with the help of simulator.
- **Simulated snoopy based cache coherence on open-source software and observed the behavior with respect to different parameters:** Simulation of snoopy coherence protocol is done on Gem5 simulator. The output is observed with respect to changing cache size, associativity, increasing cache levels.
- **Simulated directory based coherence, and protocol evolution:** Directory based coherence simulation is done on Gem5 using existing MI, MESI, MOESI, Token and Hammer protocol.
- **Modification in Gem5 source code for Simulation of benchmark:** We have done our simulation on X86 ISA, in SE mode using Splash2 benchmark. Simulation of splash2 on Gem5 in SE mode is a novel approach.

## 1.4 Thesis Structure:

This thesis contains five chapters. Chapter 1 is introduction and motivation. We then describe literature review and related work in chapter 2. Chapter 3 describe an overview of simulator. Simulation result is presented in chapter 4. Chapter 5 contain conclusion and future work.

# Chapter 2

## 2 Literature Review and related work:

Snoopy and directory methods are standard for maintaining cache coherence in multi-core systems. Both the methods have their own merits and demerits. Now-a-days we are looking for cache coherence method which satisfy bandwidth efficiency, Low-Latency Cache-to-Cache Misses and less reliability on bus like interconnect [20]. As discussed above snoopy is not good for large scalable multi-core system and directory method has directory storage overhead and high cache-to cache transfer latency. As we are using cache coherence protocol in both the above methods, so we can see protocol evolution done previously to achieve high performance irrespective of design constraint [2]. Evolution is shown through below diagram.

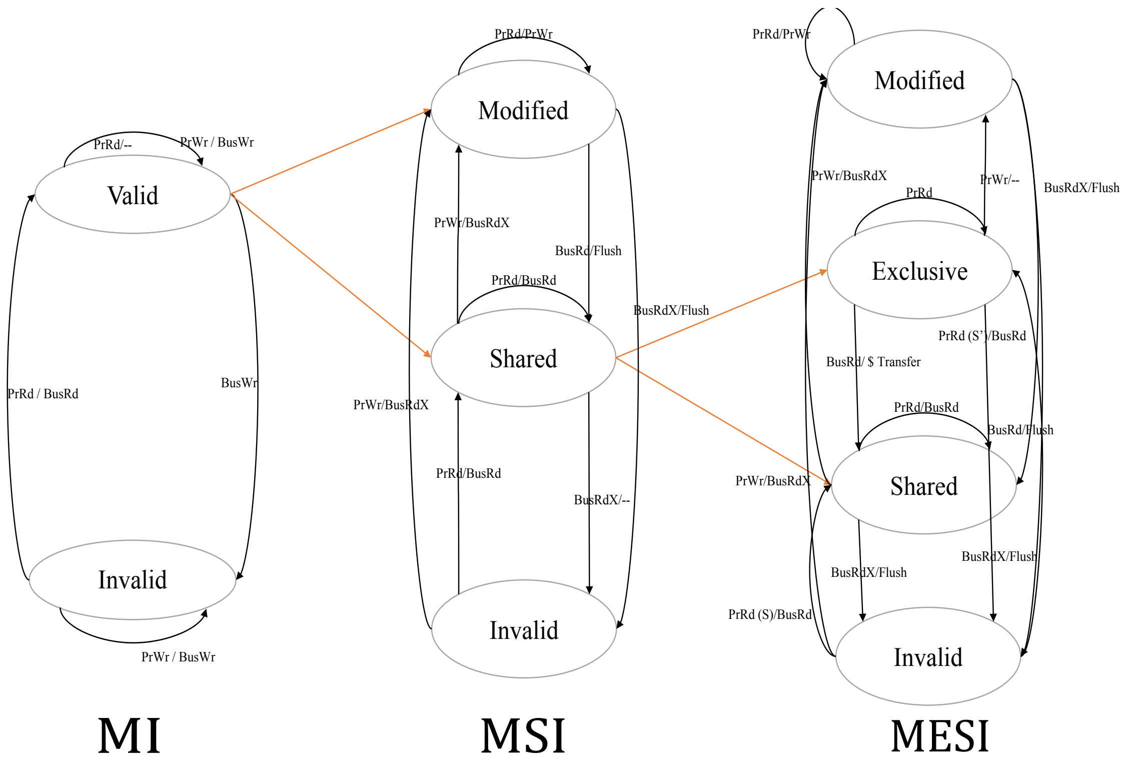


Figure 5: Evolution From MI To MSI To MESI

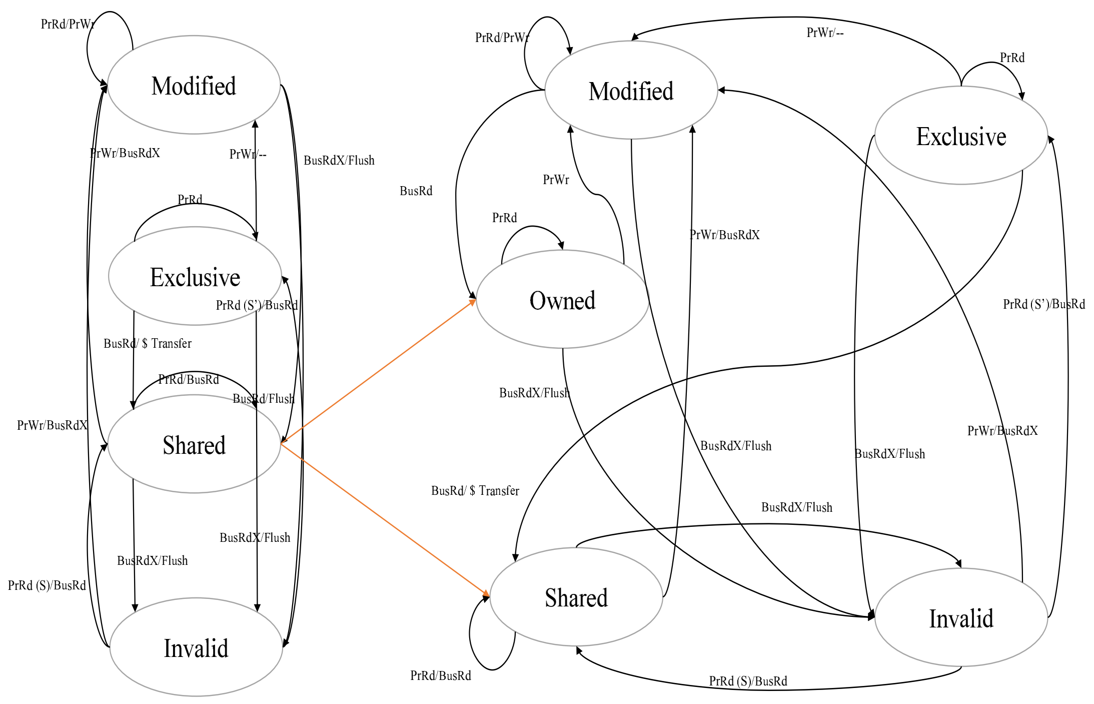


Figure 6: Evolution From MESI to MOESI

## 2.1 Protocols with advantages and disadvantages:

### 2.1.1 MI (Modified-Invalid) Protocol:

MI protocol is very basic conventional protocol used for maintaining cache coherence in shared memory multi-processor. Invalid state means neither read nor write operation can be performed. If a block is not present in cache then also it is in invalid state. If a block is in modified state then it can perform both read and write operation. During write operation only one block can be in modified state, rest are in invalid state. Below table shows MI protocol reaction to read/write/invalidate request.

| **States** | **Processors Load** | **Processors Store** | **Processors Eviction** | **Incoming Read Req.** | **Incoming Write Req** |
|---|---|---|---|---|---|
| Modified | Hit | Hit | Write-back and change to invalid | Send data | Send data and change to invalid |
| Invalid | Hit | Change to Modified | None | None | None |

Table 1: MI State Transition

- **Advantage of MI protocol:**
  1. Easy to implement.
  2. Less transient states.
  3. Fewer burdens on cache controller.
- **Disadvantage of MI protocol:**
  1. No distinction between shared and modified block.

### 2.1.2 MSI (Modified, Shared and Invalid) Protocol:

MSI was extension of MI protocol with additional **"S"** state. A block is in modified state if it is the only block in cache and modified. Block is in shared state if several copies of this block are present in other processor's cache. Invalid means either the block is absent or read/write operation is restricted. Below table shows MSI protocol reaction to read/write/invalidate request.

| States | Processor Load | Processor Store | Processor Eviction | Incoming Read Req | Incoming Write Req |
|---|---|---|---|---|---|
| Modified | Hit | Hit | Write-back and invalid | Send Data & write-back Change to Sharer | Send Data and Invalid |
| Shared | Hit | Change to modified | Silent eviction change to invalid | None | None Invalid |
| Invalid | Hit | Change to modified | None | None | None |

Table 2: State Transition Table

- **Advantage of MSI protocol:**
  1. Distinction between modified and shared state.
  2. Multiple copies of block can be present at the same time.
  3. Shared to Modify transition can be made without reading data from cache.
- **Disadvantage of MSI protocol:**
  1. On read, block goes to **"S"** state, even if it is the only copy present. This is problem because when write request arrive state changes from **"S"** to **"M"** and invalidate message is broadcast on the bus, even though it is the only copy present.
  2. Unnecessary broadcast of invalidation message.

### 2.1.3 MESI (Modified, Exclusive, Shared, and Invalid) Protocol:

After MSI we need some protocol which indicate that, this is the only cached block and is clean. Exclusive state is added to existing MSI protocol for solving the issue of unnecessary broadcast of invalidation message. Below table shows MESI protocol reaction to read/write/invalidate request.

| **States** | **Processors Load** | **Processors Store** | **Processors Eviction** | **Incoming Read Req.** | **Incoming Write Req** |
|---|---|---|---|---|---|
| Modified | Hit | Hit | Write-back and Change to Invalid | Send data Write-back and Change to Shared | Send data and Change to Invalid |
| Exclusive | Hit | Change to Modified | Silent Evict and Change to Invalid | Send data Write-back and Change to Shared | Send data Hit and Change to Invalid |
| Shared | | Change to Modified | Silent Evict and Change to Invalid | None | Change to Invalid |
| Invalid | Hit | Change to Modified | None | None | None |

Table 3: MESI State Transition

- **Advantage of MESI protocol:**
  1. Silent transition from Exclusive to Modified state.
- **Disadvantage of MESI protocol:**
  1. Extra hardware is required for deciding the transition of block when read request arrives.
  2. Every time downgrade from **"M"** state to **"S"** needs data to be written back to memory.
  3. Tradeoffs, when transition from **"M"** to **"S"** or **"I"**. Modified to shared, when data is likely to be reused and modified to invalid if data not to be reused.

### 2.1.4 MOESI (Modified, Owned, Exclusive, Shared and Invalid) Protocol:

Extension of MESI protocol, restrict write-back of data from cache to main memory. New state **"O"** is invented to avoid unnecessary write-back of data to main memory during transition from **"M"** to **"S"**. Cache block in owned state is not consistent with main memory. Below table shows MSI protocol reaction to read/write/invalidate request.

| States | Processor Load | Processor Store | Processor Eviction | Incoming Read Req | Incoming Write Req |
|---|---|---|---|---|---|
| Modified | Hit | Hit | Write-back and invalid | Send Data & Change to Owned | Send Data and Invalid |
| Owned | Hit | Change to modified | Write-back and invalid | Send Data, Write-back and Shared | Send data and Invalid |
| Shared | Hit | Change to modified | Silent eviction change to invalid | None | Change to Invalid |
| Invalid | Change to Exclusive or shared | Change to modified | None | None | None |

Table 4: MOESI State Transition Table

- **Advantage of MOESI Protocol:**
  1. Avoid extra CPU stall during write-back to main memory.
  2. One time only one cache can be owner (modified), other processors holding same block are in shared state.
- **Disadvantage of MOESI protocol:**
  1. In snoopy implementation of MOESI, when read request arrive on bus, every sharer of the block send requested data; create message contention on the bus.

#### 2.1.5 MESIF (Modified, Exclusive, Shared, Invalid, and Forward) Protocol:

MESIF protocol is used in snoopy based coherence method implementation. Earlier with MESI protocol all sharing caches respond for the read/write request from other processor that causes arrival of redundant reposes. New **"F"** state is added to the MESI protocol. This forwarding state is specialized shared state. This protocol ensures that a cache block in **"F"** state only response to the request for read/write operation [13].

#### 2.1.6 Token Coherence Protocol:

The system assigns fixed number of tokens to each block in shared memory. Number of tokens should be as large as number of processors. Initially all tokens associated with the block is hold by home module. Tokens are allowed to move throughout the system [20]. These four invariants need to be maintained for token coherence:

1. All blocks initially have T tokens.
2. A processor can perform write operation on the block only, if it holds all the tokens.
3. A processor can perform read operation on the block only, if it holds at least one token of corresponding block.
4. If a coherence message contains one or more token then it should hold data.

Mapping between token coherence protocol and MSI is shown in below table:

| **States** | **Number of Tokens** |
|---|---|
| Modified | Holding all Tokens |
| Shared | Holding 1 to T-1 Tokens |
| Invalid | Holding No Tokens |

Table 5: Mapping MSI to Token Coherence

### 2.1.7 Optimized Token Coherence Protocol:

In above token coherence technique data travel with token, which is inefficient use of bandwidth in some cases. Optimized token coherence use **"Owner"** token and extra data valid bit. These four variants need to be maintained for optimized token coherence:

1. Each block has T token including one owner token.
2. A processor can perform write operation on the block only, if it holds all the tokens.
3. A processor can perform read operation on the block only, if it holds at least one token of corresponding block and contain valid data.
4. Coherent message will contain data only if it hold owner token.

**Note:** Valid bit is set when message with data arrives including at least on token otherwise, if no token then valid bit is cleared [20].

- **Advantage of Token Coherence protocol:**

    1. Here we do not keep track of which processor is sharing block, but only count tokens associated with cached block.

### 2.1.8 Hammer Coherence Protocol:

Hammer coherence protocol used in AMD Opteron system. Hammer coherence broadcast request to bus on cache miss. If data block present on chip then requested is forwarded to all processors holding data, else memory will serve the request. Hammer coherence unlike snoopy based method do not store state corresponding to each block. Hammer coherence has disadvantage that its broadcast technique increase traffic on bus, hence power consumption [21].

## 2.2 Challenges to Cache Coherence protocol:

### 2.2.1 Snoopy Based coherence protocol:

In snoopy coherence both processor and bus lookup for cache tag at the same time, as shown in figure [7]:

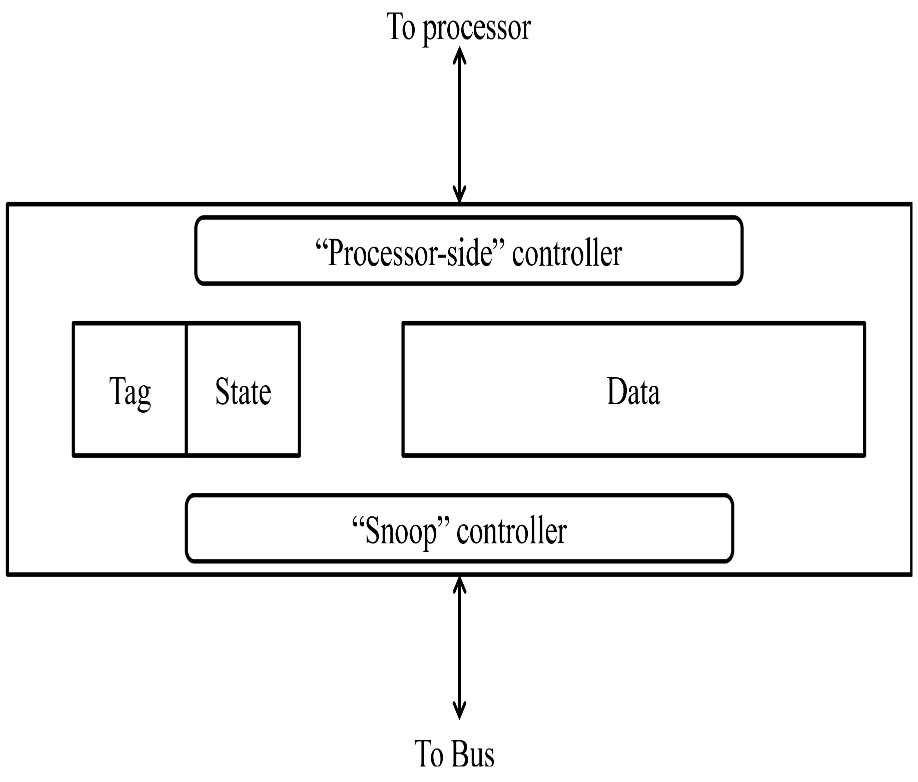


Figure 7: Simultaneous Cache Tag Lookup

There may be two possibilities; if bus receives priority then processor snoop is restricted and if processor receives high priority then snoop controller cannot respond during processor accessing cache. Solution to the above problem is implementing duplicate tags in cache as shown in figure [7]:

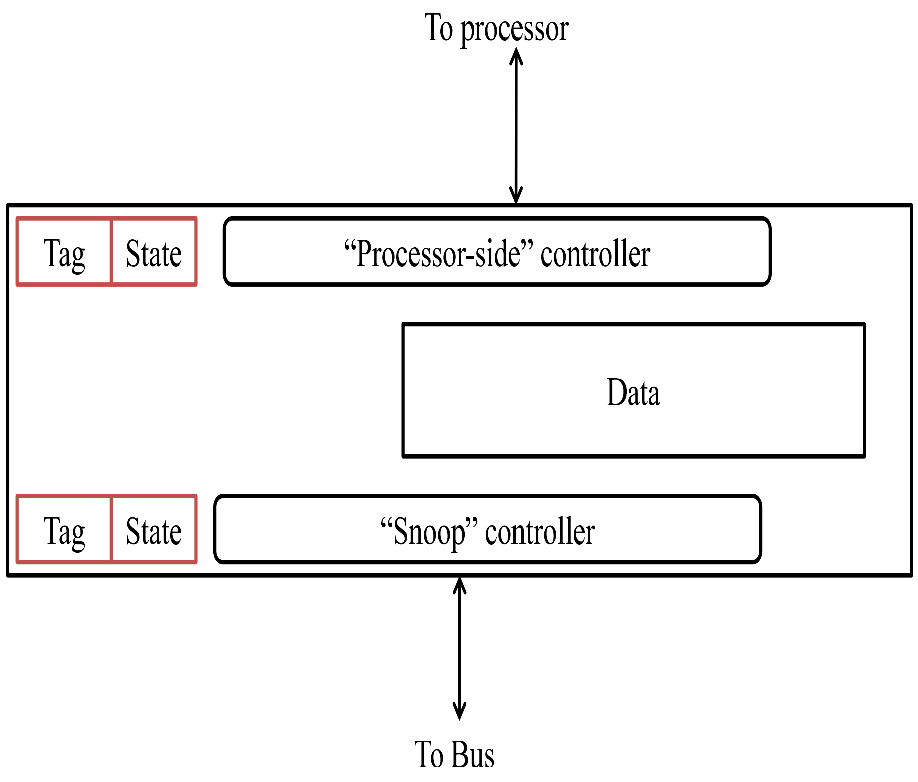


Figure 8: Duplicate Tags

Here duplicate tags need to be synchronized. But as updating tag is infrequent compare to simple lookup so less overhead for correctness.

### 2.2.2 Directory Based coherence protocol:

Directory-based coherence has 2 major challenges:

1. **Overhead of storing directory data structure:** Sparse directory technique is used to create a link list of sharers of cache block instead of maintaining array in cache as shown in figure below [7]:

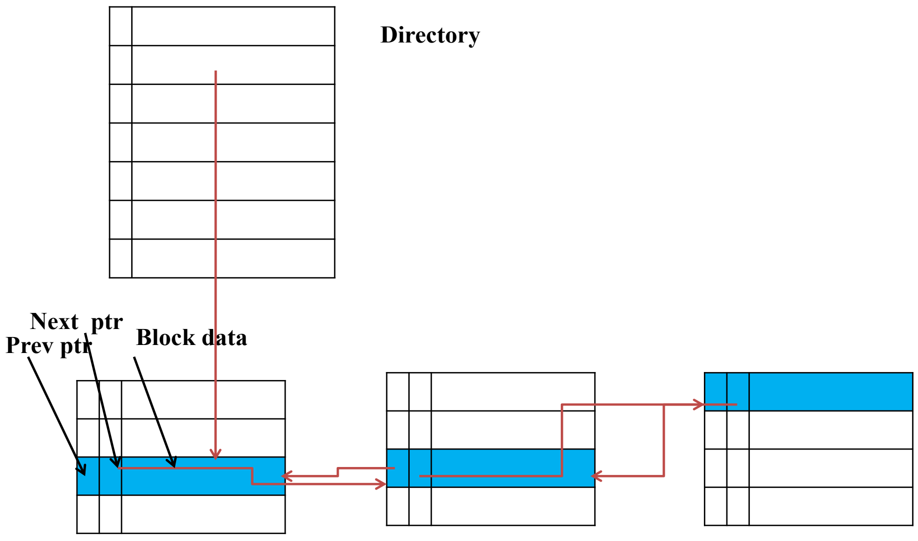


Figure 9: Link List

**On read miss:** requesting node is added to the head list.
**On write miss:** invalidation is propagated along list.
**On evict:** delete link list.

2. **Reducing request/response traffic on bus:** In normal read/write operation total five network transactions are counted as shown in figure below:

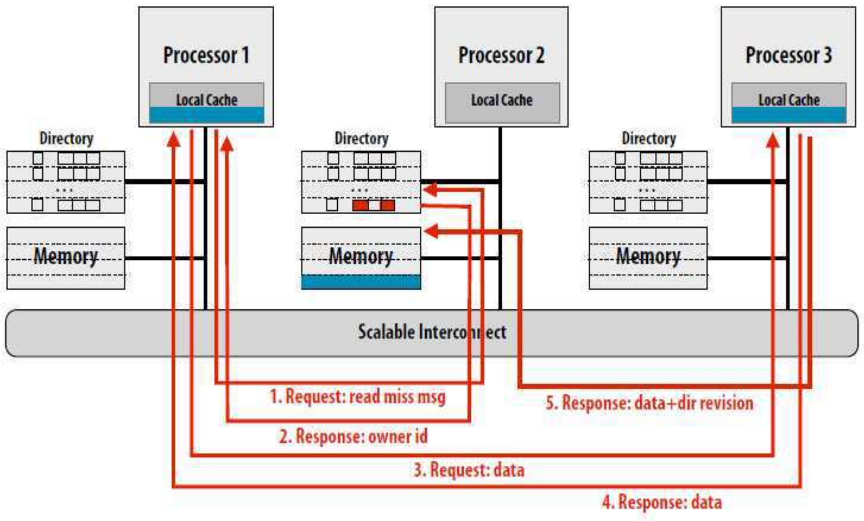


Figure 10: Network Transitions

There are two techniques namely **Intervention forwarding** and **Request forwarding** used to reduce bus traffic [7]. In intervention forwarding, on read/write miss request is sent to home directory and home directory in worst case forward request to owner node. Owner node then responds to home directory, as home directory update state and forward data to requested node. Technique of intervention forwarding is shown below:

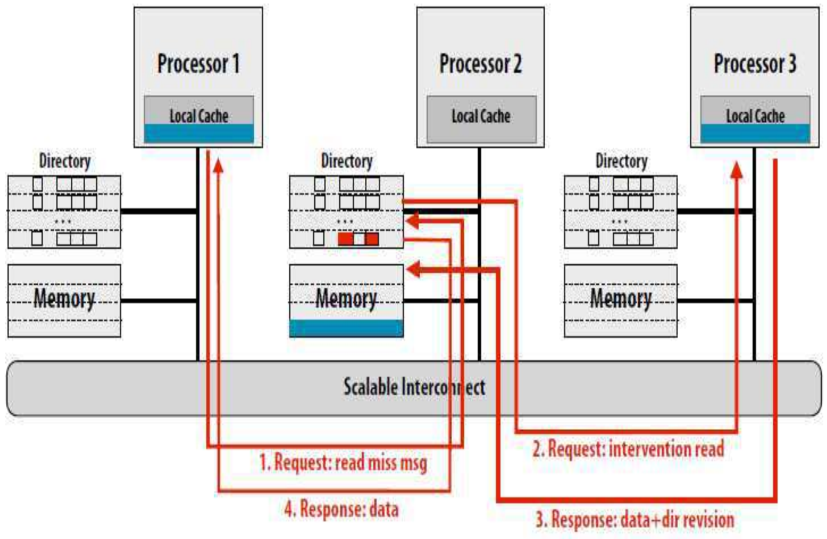


Figure 11: Network Transitions

In the above figure we can see total four transitions, one less than normal five network transitions.

Similarly, another technique called request forwarding is used to further optimize network traffic on bus. As can be seen in figure, there are only 3 transaction traffic on the bus. Transaction 3 and 4 occur parallel.

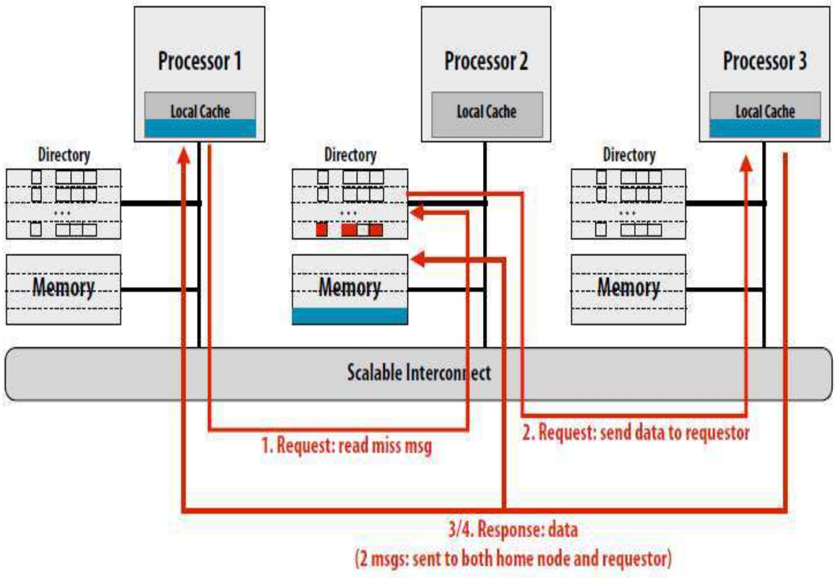


Figure 12: Network Transitions

## 2.3 Animated learning tool for cache coherence protocol:

In our thesis we have used 2 learning tools: 1) VIVIO 5.1 animation tool and 2) http://lorca.act.uji.es/ project. VIVIO 5.1 written in VC++, developed at Trinity College Dublin. Using VIVIO student can actively interact with protocol operations. An interactive animation tool respond to user input as directed. With respect to cache coherence, VIVIO helped in understanding memory access after each step. This tool can also be configurable according to user requirement like we can increases/decrease memory latency, cache block size [10, 17].

Second, flash interactive animation tool developed at Universidad Jaume I (SPAIN) covers the MSI and MESI snoopy protocol and MESI directory-based protocol. Situations covered in this animation show all possible cases that can happen in each protocol. The tool is intuitive and easy to use with no prior in depth knowledge [18].

## 2.4 Intel Nehalem Memory architecture:

Intel "Nehalem" is the nickname for the "Intel Micro-architecture", where the latter is a specific implementation of the "Intel64" Instruction Set Architecture (ISA) specification [23].

- **Cache organization in Nehalem architecture:**

Nehalem chip contain level 1 (L1) instruction and data cache and unified level 2 (L2) cache. Level 3 (L3) cache is shared among cores on chip. Level 1 cache has instruction and data cache of size 32 KB with 4-way and 8-way set associative organization respectively. Level 2 cache is write-back cache and non-inclusive unified cache. L2 cache is private to each core. Another L3 level cache is shared among cores to minimize "snoop" traffic between cores. L3 cache is inclusive meaning cache block present in L1 (instruction or data) or L2 also present in L3. Intel Core i7 follow Nehalem architecture as shown in figure below:

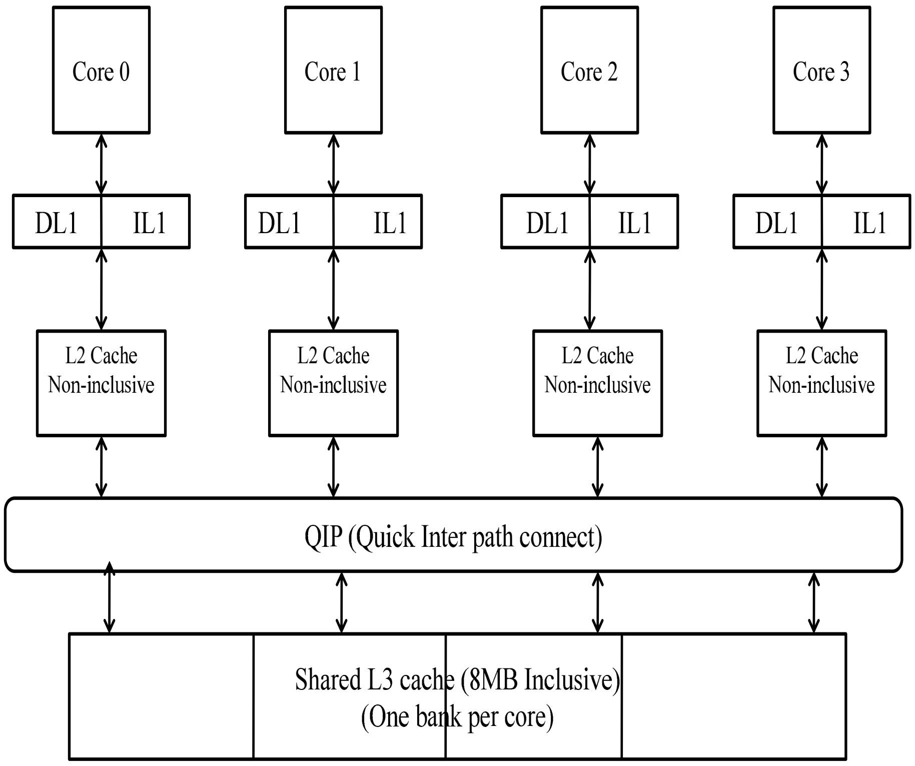


Figure 13: Nehalem Cache Architecture

Challenge with Intel 3 level cache architecture was, changes made at L1 level may not be visible to L2 level cache until it snoop to interconnect. Snoop at multi-level cache hierarchy is inefficient hence shared L3 level cache is made inclusive with respect to L1 and L2 cache. This arrangement avails state of cache block at L3 level and easy to snoop by processors.

- **Cache coherence in Nehalem architecture:**

Nehalem architecture use MESIF (Modified, Exclusive, Shared, Invalid and Forwarding) coherence protocol to maintain cache coherence with same chip caches and other chip cache via QIP [23]. MESIF is extension of well known MESI protocol. Nehalem use snoopy based method, in which processors snoop bus to observe other processors accesses to system memory. When one processor detect write request for the same block held by it then the processor invalidate its cached block. Similarly if processor modifies the cache block and request arrive for same block then processor directly gives ownership of block to requesting core.

# Chapter 3

## 3 Simulator Overview

Gem5 simulator is open-source software with all system components are configurable. Gem5 is a combination of best part of M5 and GEMS simulator. M5 supports multiple ISAs, different CPU models whereas GEMS provide flexible memory system and supports different cache coherence protocols and network connection. This simulator is widely used for study of cache coherence protocols. Gem5 simulator provide very flexible simulation environment, but we concentrate only on Memory System part. Currently Gem5 support MI, MESI_Two_Level, MESI_Three_level, MOESI, MOESI token and MOSEI hammer as coherence protocol and two memory models Classic and Ruby. Gem5 has default MOESI protocol for classic model. Classic model is fast and easily configurable with respect to Ruby, which support multiple cache coherence protocols in Gem5. Gem5 uses Domain-specific Language, SLICC, to implement variety of cache coherence protocols. A particular protocol is characterized by cache configuration, memory, DMA controller together. These components are connected through network port with defined network topology. Gem5 SLICC defines protocols as set of states, events, transitions and actions [18]. In configuration file cache characteristic, states associated, events and actions are defined. In the same configuration file transition from one state to another is written corresponding to particular event with appropriate action. If we want to devise a new protocol or make some changes in the existing protocol then we have to write our own configuration file (or we can make changes in existing configuration file) and implement it on Gem5 simulator. Initially to verify the correct working of protocol we test it with Ruby_random_tester (a pre-compiled program) and check if protocol work correctly as we are increasing workload. Once the protocol is verified we can check its performance on some standard benchmark. Gem5 has status file generated after each simulation where we can get values of required parameter like simulation time, latency, bandwidth. Gem5 ruby model facilitate to generate debug-file called "ProtocolTrace" in which we can read transition corresponding to particular event, number of cycle for the transition, for each protocol. "ProtocolTrace" is generally used to verify correctness of protocol.

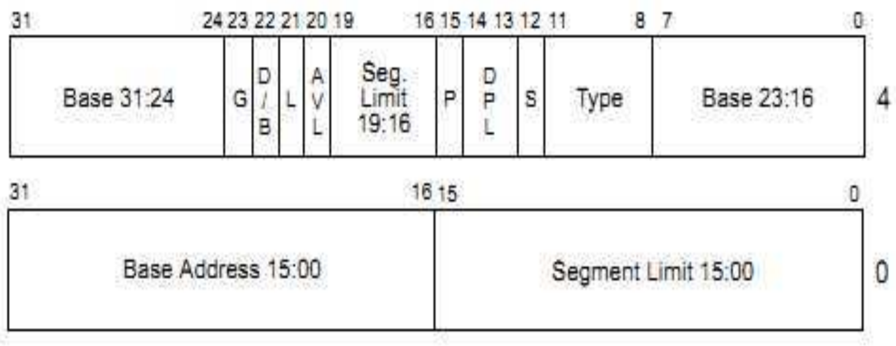
**Gem5 has two modes for simulation:** FS (Full system mode) and SE (Syscall Emulation mode). FS mode provides Linux like environment for simulation but operate very slowly. SE mode is fast and suitable, if only memory operations need to be observed.

### 3.1 ISAs present in Gem5:

Gem5 support Alpha, ARM, SPARC, MIPS, POWER and X86 instruction set architectures. In this thesis we simulated our study on X86 ISA. Figure describe X86 architecture.

### 3.2 Memory System:

Gem5 simulator has two memory models: Ruby and Classic. Classic memory model is taken from M5, whereas Ruby inherited from GEMS memory system.



| | |
|---|---|
| L | — 64-bit code segment (IA-32e mode only) |
| AVL | — Available for use by system software |
| BASE | — Segment base address |
| D/B | — Default operation size (0 = 16-bit segment; 1 = 32-bit segment) |
| DPL | — Descriptor privilege level |
| G | — Granularity |
| LIMIT | — Segment Limit |
| P | — Segment present |
| S | — Descriptor type (0 = system; 1 = code or data) |
| TYPE | — Segment type |

Figure 14: X86 Architecture

### 3.3 Basic Memory elements and Replacement Policy:

MemObjects, SimObjects, Ports and Packets. Component in Gem5 is called SimObject. MemObject is a type of SimObject. These objects communicate through Ports. Ports can be of two types: Master port, a Slave port. Master port generates request, either read/write or fetch. Slave responds to the corresponding request.

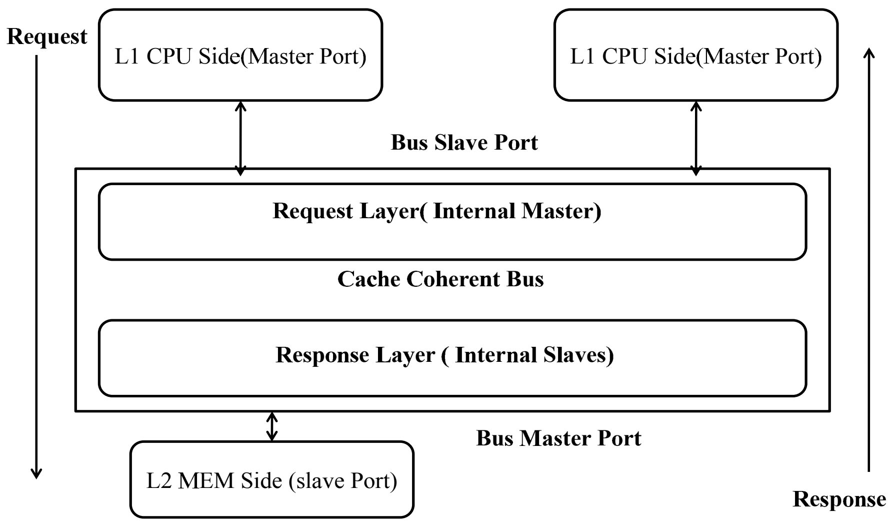


Figure 15: Communication Through Ports

LRU and Pseudo-LRU policy is used for cache block replacement in Gem5. LRU policy simply store number of times data block is accessed and victimize least recently used cache block. Pseudo-LRU policy use tree like structure for victimizing cache block and its eviction.

### 3.4 Classic Memory Model:

In classic memory model coherence is maintained using snooping protocol. Classic model has advantage of fast forwarding; simulation speed is higher as compared to Ruby. Classic memory has easy configurable components. Required memory hierarchy can be created by simply modifying python configuration. MOESI like, coherence protocol is implemented in classic memory to maintain cache coherence [3].

| MOESI States | Writeable | Dirty | Valid |
|---|---|---|---|
| Modified | 1 | 1 | 1 |
| Owned | 0 | 1 | 1 |
| Exclusive | 1 | 0 | 1 |
| Shared | 0 | 0 | 1 |
| Invalid | 0 | 0 | 0 |

Table 6: MOESI Mapped to Classic Model in Gem5

In classic memory model bus have two type packets one is memory-mapped and other is snooping packets. Memory-mapped request go vertically from cache to memory and response arrives from memory to cache. Snooping packets move horizontally from cache to cache and down from cache to memory. Gem5 classified snoop packets as normal snoop and express snoop packets on the basis of snoop packets movement.

### 3.5 Ruby Memory Model:

Ruby system is slower as compared to classic memory model as ruby provide more flexible memory infrastructure for simulating different memory system. Ruby uses SLICC (Specification Language for Implementing Cache Coherence), a domain specific language to define many cache coherence protocols. Overall protocols combine Cache, Memory and DMA controllers written in SLICC. Control logic of protocol is written in C++, incorporated for state transition. Communication in ruby memory system is done via RubyPort and MessageBuffer. Ruby has advantage of flexible cache coherence protocol.

### 3.6 Memory Request Flowchart in Ruby:

Below flowchart describe how request flow through Ruby and different associated components. Process is described in detail:

1. A memory request from core, in form of M5 packets arrives inside Ruby system. Ruby port convert this packet to RubyRequest object, which can be easily recognize by Ruby components.

2. I request is for memory access not for I/O, and then request is passed to Sequencer. Each sequencer object is attached with one core. Sequencer does accounting and request allocation for the request and finally request is enqueue to mandatory queue. If cache hierarchy is multilevel then top level cache controller (L1 cache) dequeue request from mandatory queue and push it to next level.

3. If requested cache block is present in L1 cache then check for coherence permission (probably for transient state), then request is serviced to requesting core.
4. All request latency is accounted and send back to requesting core.

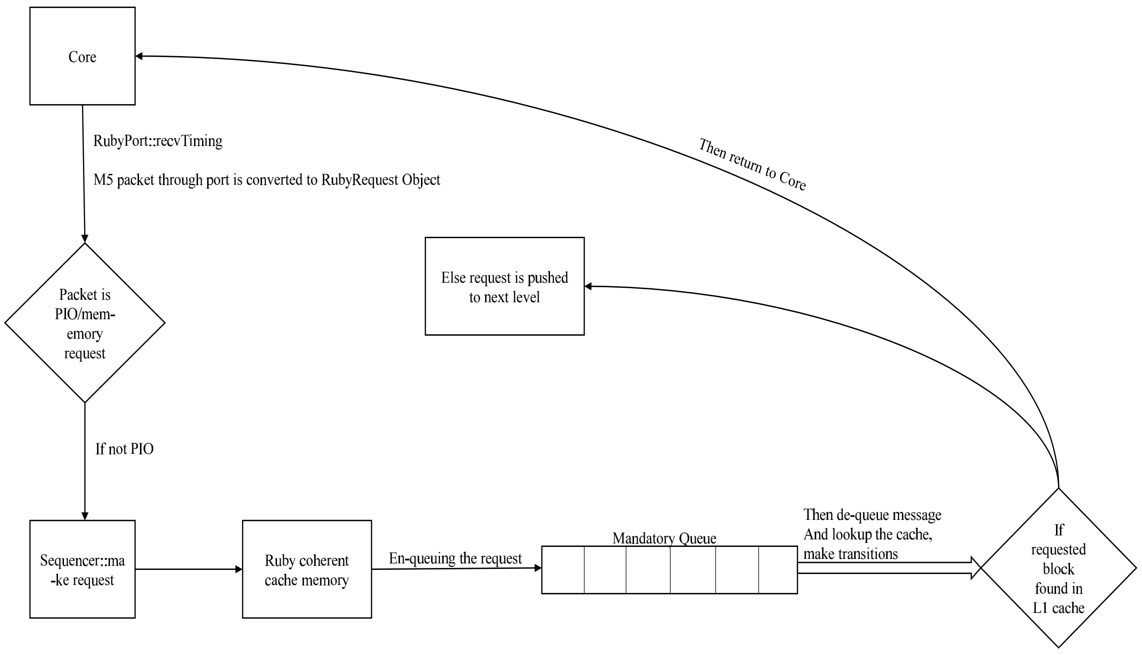

Core
RubyPort::recvTiming
M5 packet through port is converted to RubyRequest Object
Then return to Core
Else request is pushed to next level
Packet is PIO/mem-emory request
If not PIO
Sequencer::ma -ke request
Ruby coherent cache memory
En-queuing the request
Mandatory Queue
Then de-queue message And lookup the cache, make transitions
If requested block found in L1 cache

Figure 16: Request Flowchart For Single Core

## 3.7 Coherence Messages

| Messages | Description |
|---|---|
| ACK/NACK | Positive/Negative acknowledgement for request. Example, Write-back. |
| GETS | Request for Shared. |
| GETX | Request for Exclusive Access. |
| INV | Invalidation message sent when writing to particular block. |
| PUTX | Request for write-back of cache block in Exclusive state. |
| PUTS | Request for write-back to cache in Shared state. |
| PUTO | Request for write-back to cache in Owned state. |
| PUTO_Sharers | Write-back in Owned state but other Sharer of block exist. |
| UNBLOCK | Message to UNBLOCK. |

Table 7: Coherence Messages

## 3.8 Finite State Machine:

FSM in SLICC composed of events, state (stable and transient), action and transition. An event is input to the state machine corresponding to arrival of particular message. After receiving event system trigger some action relevant to the event. State transition is atomic. States are defined using state_declaration keyword with appropriate Access Permission. A transition is defined by initial state going to final state/transient state with corresponding action performed. Cache controller controls all the transitions and blocks the request until previous transition is completed. For example: Transition for invalid state to transient state as given below.

```
Transition (I, Load, Ifetch, IS)
v_allocateTBE;
i_allocateL1CacheBlock;
a_issueRequest;
p_profileMiss;
m_popMandatoryQueue;
```

### 3.9 Cache Coherence Protocol Supported by Ruby:

Following are the cache coherence protocol supported by Ruby memory model:

| **Coherence Protocols** | **Description** |
|---|---|
| MI | 1 Level Cache |
| MESI_Two_Level | 2 Level Caches, Inclusive hierarchy |
| MESI_Three_Level | 3 Level Caches, Inclusive hierarchy |
| MESI_CMP_Directory | 2 Level Caches private caches, non-inclusive hierarchy |
| MOESI_CMP_Token | 2 Level Caches |
| MOESI_Hammer | 2 Level private Caches strictly exclusive |

Table 8: Coherence Protocols supported by Ruby

### 3.10 Protocol Independent Memory Components:

There are four independent components Sequencer, Cache Memory, Cache Replacement policy and Memory controller. The sequencer is responsible for requesting read/write/atomic operation from processor. There is one sequencer for each core. Cache memory is set associative with adjustable parameters like associativity, cache size, and replacement policy.

### 3.11 Interconnection Network in Gem5:

Ruby supports two network models: **"Simple"** and **"Garnet"**. Simple network is by default present whereas Garnet network is more detailed and slower than simple network. The **simple network** models hop-by-hop network traversal, but abstracts out detailed modeling within the switches. The flow-control is implemented by monitoring the available buffers and available bandwidth in output links before sending.
**Garnet** is a detailed interconnection network model inside gem5. It consists of a detailed fixed-pipeline model, and an approximate flexible-pipeline model. Garnet has features, like determinstic routing using routing tables, variable link bandwidth and multi-cast message delivered as multiple uni-cast messages at the network interface [3].

# Chapter 4

## 4 Implementation and Simulation Result

we have done simulation for both Classic and Ruby model of Gem5. As we know from above chapter 3, that Snoopy coherence method is supported by Classic memory model and Directory coherence method is supported by Ruby memory model. In below sections we discuss about simulation setup, benchmark and simulation environment.

### 4.1 Simulator Setup

Following are steps for simulator setup:

1. Install Linux 64 bit operating system.
2. Open terminal, write below command in command line.

   **"sudo apt-get install mercurial scons swig gcc m4 python python-dev libgoogle-perftools-dev g++"**

3. After installation of above packages, write command for gem5 installation in command line as:

   **"hg clone http://repo.gem5.org/gem5"**

4. After gem5 installation, change directory to gem5 using:

   **"cd gem5"**

5. Now build gem5 binary "gem5.opt" for simulation. We can choose X86, ARM, or ALPHA as ISA for simulation.

   **"scons build/X86/gem5.opt"**

6. After step 5 gem5 installation is complete. We can test proper installation by running a test program like **"hello.c"** present in gem5 package, using command as:

   **"build/X86/gem5.opt configs/example/se.py -c tests/test-progs/hello/bin/x86/linux/hello"**

### 4.2 Simulation Environment

Gem5 has 2 modes for simulation SE and FS. SE is faster as compare to FS mode. In this thesis we have used SE mode. Among available ISAs in Gem5 we worked with X86 architecture.

## 4.3 Benchmark Used

Splash-2 benchmark is used for simulation study [6]. Splash-2 parallel applications are used to study behavior of shared-address-space multiprocessors. Splash-2 suite contains application programs that are suitable for study of coherence protocols. Parallel application **"fmm"** and **"ocean"** of Splash-2 suite is used for simulating shared memory multiprocessor system [5, 25].

## 4.4 Running parallel program on Gem5 in SE mode:

To run parallel program on Gem5 in SE mode we need M5thread library. First we statically compile our program with the help of M5thread suite then run compiled program code on Gem5 simulator. Following commands are used for successful simulation.

1. Change to the directory where we want M5thread installation.
2. Use **" hg clone http://repo.gem5.org/m5threads"** to download M5thread library.
3. Now change directory to m5thread as, "cd m5thread".
4. Use command **"make"** to compile libpthread.
5. Now run a test program, present in test package.
6. To run our own parallel program or parallel benchmark we need to do modification in **"makefile"** of m5thread after which use **"make"** command to compile program.
7. Now again change the directory to gem5 and simulate the program (give path of compiled program) using below command.

   **"./build/X86/gem5.opt configs/example/se.py -n 4 -c <u>m5thread path</u> "**

## 4.5 Snoopy Coherence Method

Gem5 provide classic memory model for snoopy coherence protocol simulation. Classic memory only supports MOESI like coherence protocol.

- **Architecture support:** We simulate single level and double level cache hierarchy, each with varying parameters like cache line size, associativity and cache size.
- **Compiled binary used for snoopy coherence:** We use updated **"MemTest"** for observing our simulation result. MemTest generate random read/write request hence facilitate true sharing. We can configure cache block size, associativity and L1,L2 cache size according to our requirement in **"memtest.py"**. Cache level hierarchy can also be set using **"-treespec"** parameter present in "memtest.py". For example:

  1. **"4:1"** means four L1 level caches, connected separately to four cores. Main memory connected to L1 caches through **"cpu-side-bus"**.

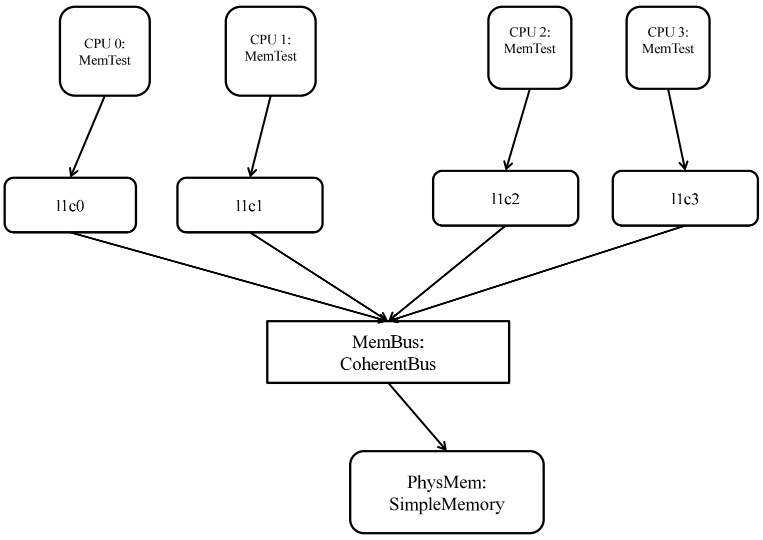


Figure 17: “4:1” System

2. **“3:2:1”** means three L2 level caches, each connected to two L1 level caches and finally each L1 level connected with separate cores.

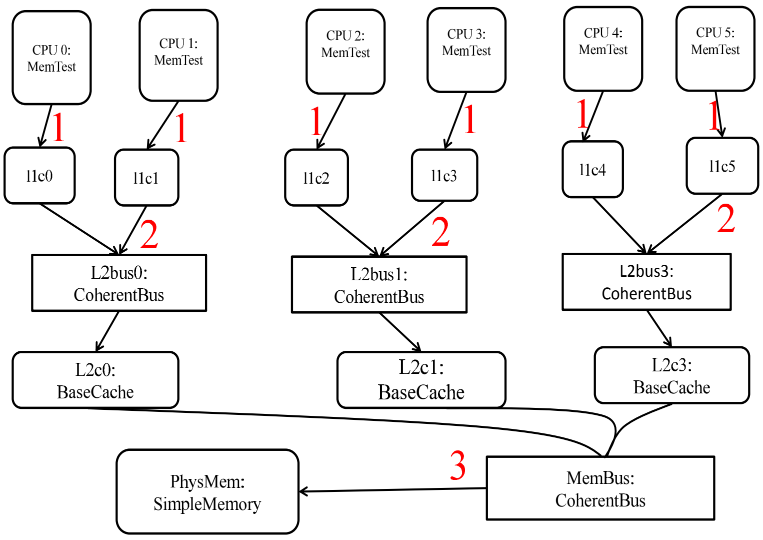


Figure 18: “3:2:1” System

- **Metrics for performance analysis:** For performance analysis we use following parameters:

  1. Test execution time and CPU memory references:
  2. Total data through bus and total snoop data through bus.
  3. Snoop layer utilization.
  4. And, Bus throughput.

- **MemTest setup:** MemTest is considered as small scale workload to evaluate snoopy coherence protocol based on above performance metrics. We first did experiment with **"4:1"** single level cache architecture to observe performance metrics value. Next, we configure **"3:2:1"** double level cache architecture. MemTest system **"4:1"** will have four CPU, generate random read/write request. Each CPU has private L1 cache of 32KB (default, change for our experiment) with associativity 4. Default cache line size is 64 bytes which we configure (increase/decrease) for our experiment. L1 level caches are connected through common coherent bus and share snoop data through this bus. Whereas in **"3:2:1"** system we have L1 and L2 level caches, in which snoop data is shared at both level through coherent bus (as shown in figure 17 above).

- **Simulation result:** Result is shown with the help of following tables and bar graph. Table 9 is simulation for "4:1" system means four L1 level caches, connected separately to four cores and main memory connected to L1 caches through **"cpu-side-bus"**, for fixed block size= 64 bytes and associativity= 4. Table 10 is simulation for "4:1" system for fixed L1 cache size= 32 Kbytes and associativity= 4. Pictorial behavior of coherence traffic is shown through bar graph in figure 1 and figure 2. The behavior of table 1 is justified as increase in coherence traffic occurs because the probability of a miss being caused by an invalidation increase with cache size, since fewer entries are bumped due to capacity [15]. Similarly, behavior of table 2 is justified as increase misses due to false sharing and due to fetching unnecessary data. Another reason can be due to conflict in fixed-size cache [15]. Table 11 (block size= 64b, L1 cache size=32Kb) shows how snoopy data become bottleneck on increasing number of cores with respect to Snoopy coherence protocol.

| **L1 cache Size (Kbytes)** | **Simulation time(ticks)** | **Snoopy data (bytes)** | **Data through bus (bytes)** | **Throughput (bytes/sec)** | **Write-back/ Memory references** |
|---|---|---|---|---|---|
| 16 | 308960 | 514048 | 4067008 | 14827343345 | 17300 |
| 32 | 230672 | 972032 | 2865472 | 16636193383 | 12672 |
| 64 | 126640 | 1580160 | 1178624 | 21784459886 | 5251 |
| 128 | 91659 | 1803008 | 254272 | 2244493175 | 0 |

Table 9: Varying L1 cache size

| **Cache Block Size (bytes)** | **Simulation time(ticks)** | **Snoopy data (bytes)** | **Data through bus (bytes)** | **Throughput (bytes/sec)** | **Write-back/ Memory references** |
|---|---|---|---|---|---|
| 16 | 235519 | 223824 | 729760 | 4048862300 | 11214 |
| 32 | 229220 | 478592 | 1451008 | 8418113603 | 12350 |
| 64 | 230672 | 972032 | 2865472 | 16636193383 | 12672 |
| 128 | 247878 | 1996032 | 5722240 | 31137382099 | 13001 |

Table 10: Varying cache block size

| No. of cores | Simulation Time | Total Snoopy Data |
|---|---|---|
| 4 | 230672 | 972032 |
| 8 | 232798 | 3893952 |
| 16 | 405482 | 12044288 |
| 32 | 792691 | 27811264 |

Table 11: Snoop data and No. of cores

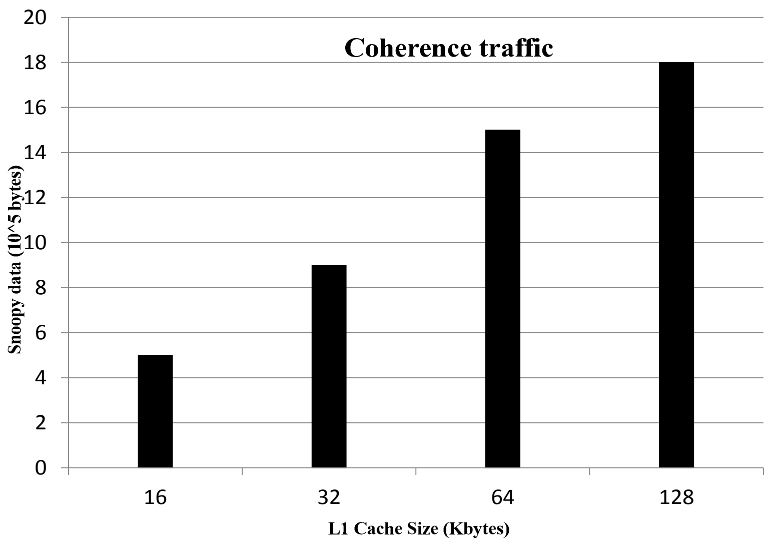


Figure 19: Varying L1 cache size

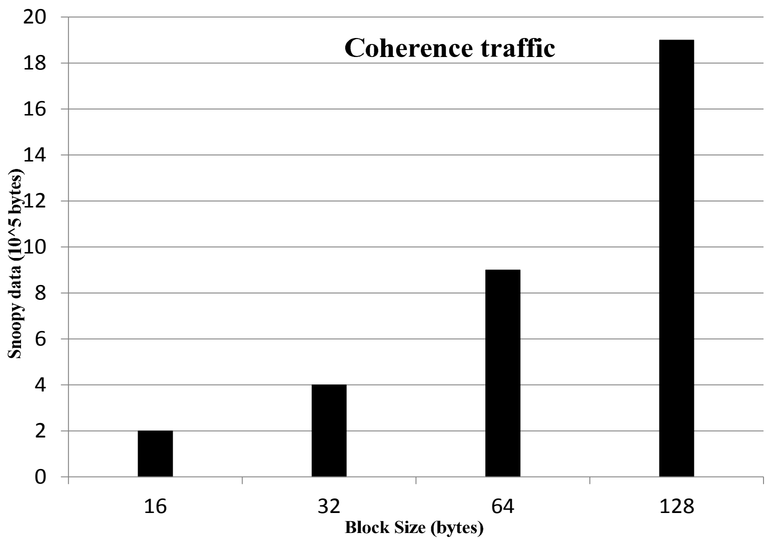


Figure 20: Varying cache block size

## 4.6 Directory Coherence Technique

Directory coherence technique is implemented with the help of Ruby model of Gem5 simulator.Ruby uses SLICC (Specification Language for Implementing Cache Coherence), a domain specific language to define many cache coherence protocols. Overall protocols combine Cache, Memory and Dma controllers written in SLICC. Control logic of protocol is written in C++, incorporated for state transition. Communication in ruby memory system is done via RubyPort and MessageBuffer. Ruby has advantage of flexible cache coherence protocol. Ruby system support above mentioned directory coherence protocols. Here we will show stable state and transient state of each protocol at different cache level. Transient states are used to maintain atomic transitions.

- **MI Coherence protocol:** It is a simple cache coherence protocol with 1-level private cache. There are only 2-stable states that is M(modified) and I(invalid). Transient states II, MI, MII, IS, IM support atomic transitions [9].
- **MESI Coherence protocol:** This protocol models 2-level cache. L1 cache is private to core and L2 cache is shared. Below table show states present at L1 cache controller [4, 12].

| **States** | **Description of the state** |
|---|---|
| M | Cache line is exclusively held in L1 cache. That data is only legal copy in system. |
| E | The cache block is read from memory with exclusive permission. |
| S | The cache block in shared state held by 1 or more L1 caches and/or by the L2 cache. |
| I / NP | Cache block is either invalid or not present. |
| IS | It is a transient state. The cache block is waiting for response after issuing read request. |
| IM | A transient state. The cache block is waiting for response after issuing write request. |
| SM | A transient state. Request is waiting for acknowledgement after issuing exclusive write request when same block is in “S” state. |
| IS_I | It is a transient state. This means that while in IS state the cache controller received Invalidation from the L2 Cache’s directory. |
| M_I | It’s a transient state. Awaiting response from L2 cache for write-back of cache block when block get replaced. |
| SINK_WB_ACK | A transient state. State is reached when write-back lost the race against forward request from another core. |

Table 12: States at L1 cache of MESI coherence model

The L2 cache block encodes the information about the status and permissions of the cache blocks in the L2 cache as well as the coherence status of the cache block that may be present in one or more private L1 caches.

- **MOESI Coherence protocol:** Coherence protocol contain Owned state to optimize MESI. Below table show states at L1-cache level [8, 12].

| **States** | **Description** |
|---|---|
| MM | Exclusively held cache block and about to modify. |
| MM_W | Exclusively held cache block and about to modify, transform to "MM" after some time. |
| O | Only one block in "O" state. Rest same cache block are in shared state. |
| M | Exclusively held block. Only single copy is presnt in private cache. |
| M_W | Exclusively held block. On silent eviction change to "MM_W" state. |
| S | Shared state of cache block. Held by one or more cache. |
| I | cache block is invalid or not present. |

Table 13: MOESI L1-cache States

- **MOESI Token Coherence protocol:** This protocol models 2-level cache hierarchy. It maintains coherence explicitly exchanging and counting tokens. Fix number of token are assigned to each cache block in the beginning, the number of token remains unchanged. State table of cache coherence at different level [12, 24].

| **States** | **Description** |
|---|---|
| MM | Exclusively held cache block and about to modify. |
| MM_W | Exclusively held cache block and about to modify, transform to "MM" after some time. |
| O | Only one block in "O" state. Rest same cache block are in shared state. |
| M | Exclusively held block. Only single copy is presnt in private cache. |
| M_W | Exclusively held block. On silent eviction change to "MM_W" state. |
| S | Shared state of cache block. Held by one or more cache. |
| I | cache block is invalid or not present |

Table 14: L1-level Cache

| **States** | **Description** |
|---|---|
| NP | Exclusively held cache block and about to modify. |
| O | Only one block in "O" state. Rest same cache block are in shared state. |
| M | Exclusively held block. Only single copy is presnt in private cache. Stores are not allowed in this state. |
| S | Shared state of cache block. Held by one or more cache. |
| I | cache block is invalid or not present. |

Table 15: L2-level cache

| **States** | **Description** |
|---|---|
| O | Owner of the block. |
| NO | Ownership is not held by this. |
| L | Temporarily locked. |

Table 16: Directory Controller

- **MOESI Hammer Coherence Protocol:** Its a 2-level coherence protocol with private L1 and L2 exclusive cache. States in this coherence protocol at different level are: [12, 14]

| States | Description |
|---|---|
| MM | Exclusively held cache block and about to modify. |
| O | Only one block in "O" state. Rest same cache block are in shared state. |
| M | Exclusively held block. Only single copy is presnt in private cache. Stores are not allowed in this state. |
| S | Shared state of cache block. Held by one or more cache. This is most updated cache block. |
| I | cache block is invalid or not present. |

Table 17: Cache controller states

| States | Description |
|---|---|
| NX | Not Owner, probe filter entry exists, block in O at Owner. |
| NO | Not Owner, probe filter entry exists, block in E/M at Owner. |
| S | Data clean, probe filter entry exists pointing to the current owner. |
| O | Data clean, probe filter entry exists. |
| E | Exclusive Owner, no probe filter entry. |

Table 18: Directory Controller

- **MESI Three_level Coherence Protocol:** This coherence model has same state description as MESI_Two_Level but less number of transient states in L0 and L1-cache.

#### 4.6.1 Verification and Simulation of Ruby coherence protocol:

Gem5 Ruby model has precompiled "sequential-test" and "random-test" program to verify correct working of protocol. Ruby model also has "–debug-flag" option with multiple values to trace correct working of protocol. For example, below command is used for protocol trace.

**"./build/X86/gem5.opt –debug-flag=ProtocolTrace configs/example/ruby_random_test.py"**

For verification and simulation we have number of input parameters that can be set in accordance with architectural requirement. Some of the frequently used options are:

- **Type of CPU:** Ruby model has Simple, Detailed and Atomic type of CPU. For using proper CPU type we should refer status matrix of gem5. Default is "atomic" in gem5.
- **Number of CPU/cores:** Number of cores can be increased/decreased for testing protocol. Number of L1 caches are equal to number of cores. Default number is "1" in gem5.
- **Memory size:** This is size of physical memory, sometime need to be increased for running benchmark programs. Default size is "512MB" in gem5.
- **Number of directory controllers:** Directory controller attach with each core handle communication between cores, core and memory. Default number is "1" in gem5.

- **Number of L1 and L2 caches:** Number of L1-cache is equal to number of CPU and number of L2-cache is 1 in shared L2 cache but private to each core in three level cache hierarchy. Default number is "1" for L2-cache in gem5.
- **Size of L1, L2 and L3 caches:** L1-cache has separate instruction and data cache. L2 and L3 caches are unified. Size of caches can be varied for optimized configuration. Default "l1d_size" is "64kB", "l1i_size" is "32kB", "l2size" is "2MB" and "l3size" is "16MB" in gem5.
- **Cache Associativity:** Cache associativity decide number of blocks present in set-associative cache. Default "l1d_assoc" is "2", "l1i_assoc" is "2", "l2_assoc" is "8" and "l3_assoc" is "16" in gem5.
- **Cache Block_size:** Cache block size varied if we want to increase/decrease number of blocks in cache. Default cache block_size is "64b" in gem5.
- **Number of Loads:** Number of loads is input of testing protocol.

Below table show simulation of coherence protocols using precompiled "ruby_random_tester" program on 8 core system with "10000" workload on Gem5. ProtocolTrace and RubyNetwork "–debug-flag" is used to monitor correct transition of states and network traffic respectively. MI protocol is simple general coherence protocol. Further, we will compare MESI 2-level coherence protocol with MESI 3-level and MOESI_CMP_directory with MOESI_CMP_token.

| **Protocol** | **Total execution time (sec)** | **ProtocolTrace** |
|---|---|---|
| MI 1-level coherence protocol | .002580 | Stable state and transient state transitions at L1-cache level. |
| MESI 2-level coherence protocol | .011705 | Stable state and transient state transitions at L1 and L2-cache level. |
| MESI 3-level coherence protocol | .012051 | Stable state and transient state transitions at L1, L2 and L3-cache level. |
| MOESI_CMP_directory | .003499 | Optimized "Owned" state at L1 and L2-cache level. |
| MOESI_CMP_token | .002223 | State transitions using token counting. |
| MOESI_hammer | .002956 | Same stable state and transient state at both cache levels |

Table 19: "ruby_random_test" experiment

Further we compare MESI_Two_level and MESI_Three_level performance using "fft", "lu" and "radix" program code of splash-2 benchmark on 4-processors system. We can see MESI_Three_level take less time than MESI_Two_level because latency due to main memory access is reduced in three level hierarchy. Table 20 (a) and (b) show comparison for "radix". Table 21 (a) and (b) show comparison for "fft". Similarly, Table 22 (a) and (b) show comparison for "lu".

| **Processor** | **Total time** | **Rank time** | **Sort time** |
|---|---|---|---|
| P0 | 3782 | 3129 | 373 |
| P1 | 3782 | 3256 | 371 |
| P2 | 3782 | 3255 | 372 |
| P3 | 3781 | 3390 | 372 |
| Avg | 3782 | 3258 | 372 |
| Min | 3781 | 3129 | 371 |
| Max | 3782 | 3390 | 373 |

Table 20: (a) "radix" simulation on MESI_Two_level system for 2048 keys

| **Processor** | **Total time** | **Rank time** | **Sort time** |
|---|---|---|---|
| P0 | 2114 | 1769 | 211 |
| P1 | 2112 | 1830 | 209 |
| P2 | 2113 | 1831 | 212 |
| P3 | 2115 | 1889 | 209 |
| Avg | 2114 | 1830 | 210 |
| Min | 2112 | 1769 | 209 |
| Max | 2115 | 1889 | 212 |

Table 20: (b) "radix" simulation on MESI_Three_level system for 2048 keys

| **Processor** | **Computation time** | **Transpose time** |
|---|---|---|
| P0 | 2036 | 304 |
| P1 | 2037 | 308 |
| P2 | 2038 | 313 |
| P3 | 2040 | 315 |
| Avg | 2038 | 310 |
| Max | 2040 | 315 |
| Min | 2036 | 304 |

Table 21: (a) "fft" simulation on MESI_Two_level system

| **Processor** | **Computation time** | **Transpose time** |
|---|---|---|
| P0 | 1048 | 159 |
| P1 | 1049 | 161 |
| P2 | 1049 | 163 |
| P3 | 1049 | 165 |
| Avg | 1049 | 162 |
| Max | 1049 | 165 |
| Min | 1048 | 159 |

Table 21: (b) "fft" simulation on MESI_Three_level system

| **Processor** | **Total time** | **Diagonal time** | **Perimeter time** | **Interior time** | **Barrier time** |
|---|---|---|---|---|---|
| P0 | 7669 | 624 | 1062 | 3362 | 2617 |
| P1 | 7666 | 2 | 1981 | 3360 | 2320 |
| P2 | 7669 | 621 | 793 | 3363 | 2888 |
| P3 | 7666 | 4 | 1715 | 1686 | 4260 |
| Avg | 7668 | 313 | 1388 | 2943 | 3021 |
| Min | 7666 | 2 | 793 | 1686 | 2320 |
| Max | 7669 | 624 | 1981 | 3363 | 4260 |

Table 22: (a) "lu" simulation on MESI_Two_level system for 64*64 matrix

| **Processor** | **Total time** | **Diagonal time** | **Perimeter time** | **Interior time** | **Barrier time** |
|---|---|---|---|---|---|
| P0 | 3897 | 318 | 539 | 1709 | 1329 |
| P1 | 3897 | 0 | 1009 | 1713 | 1173 |
| P2 | 3897 | 311 | 404 | 1701 | 1478 |
| P3 | 3897 | 1 | 870 | 860 | 2164 |
| Avg | 3897 | 158 | 706 | 1496 | 1536 |
| Min | 3897 | 0 | 404 | 860 | 1173 |
| Max | 3897 | 318 | 1009 | 1713 | 2164 |

Table 22: (b) "lu" simulation on MESI_Three_level system for 64*64 matrix

In below table comparison of directory coherence protocol and token coherence protocol is shown. Token coherence protocol is a novel approach and perform better than directory coherence protocol. Token coherence avoid need for total ordering of messages on network which in other words reduce additional latency [11]. Table 23 (a) and (b) show comparison for "radix". Table 24 (a) and (b) show comparison for "fft". Similarly, Table 25 (a) and (b) show comparison for "lu".

| **Processor** | **Total time** | **Rank time** | **Sort time** |
|---|---|---|---|
| P0 | 3492 | 3016 | 187 |
| P1 | 3496 | 3154 | 186 |
| P2 | 3494 | 3144 | 187 |
| P3 | 3496 | 3287 | 186 |
| Avg | 3494 | 3150 | 186 |
| Min | 3492 | 3016 | 186 |
| Max | 3496 | 3287 | 187 |

Table 23: (a) "radix" simulation on MOESI_CMP_directory system for 1024 keys

| Processor | Total time | Rank time | Sort time |
|---|---|---|---|
| P0 | 2645 | 2284 | 140 |
| P1 | 2643 | 2384 | 140 |
| P2 | 2647 | 2382 | 140 |
| P3 | 2644 | 2485 | 141 |
| Avg | 2645 | 2384 | 140 |
| Min | 2643 | 2284 | 140 |
| Max | 2647 | 2485 | 141 |

Table 23: (b) "radix" simulation on MOESI_CMP_token system for 1024 keys

| Processor | Computation time | Transpose time |
|---|---|---|
| P0 | 2041 | 306 |
| P1 | 2040 | 309 |
| P2 | 2041 | 313 |
| P3 | 2036 | 315 |
| Avg | 2040 | 311 |
| Max | 2041 | 315 |
| Min | 2036 | 306 |

Table 24: (a) "fft" simulation on MOESI_CMP_directory system

| Processor | Computation time | Transpose time |
|---|---|---|
| P0 | 1535 | 228 |
| P1 | 1533 | 231 |
| P2 | 1535 | 234 |
| P3 | 1534 | 237 |
| Avg | 1534 | 232 |
| Max | 1535 | 237 |
| Min | 1533 | 228 |

Table 24: (b) "fft" simulation on MOESI_CMP_token system

| Processor | Total time | Diagonal time | Perimeter time | Interior time | Barrier time |
|---|---|---|---|---|---|
| P0 | 7673 | 624 | 1061 | 3362 | 2623 |
| P1 | 7671 | 2 | 1981 | 3363 | 2321 |
| P2 | 7672 | 622 | 792 | 3363 | 2891 |
| P3 | 7670 | 1 | 1714 | 1686 | 4265 |
| Avg | 7672 | 312 | 1387 | 2944 | 3025 |
| Min | 7670 | 1 | 792 | 1686 | 2321 |
| Max | 7673 | 624 | 1981 | 3363 | 4265 |

Table 25: (a) "lu" simulation on MOESI_CMP_directory system for 64*64 matrix

| **Processor lu** | **Total time** | **Diagonal time** | **Perimeter time** | **Interior time** | **Barrier time** |
|---|---|---|---|---|---|
| P0 | 5756 | 468 | 797 | 2521 | 1968 |
| P1 | 5753 | 1 | 1485 | 2522 | 1742 |
| P2 | 5759 | 467 | 594 | 2523 | 2172 |
| P3 | 5762 | 0 | 1285 | 1265 | 3209 |
| Avg | 5758 | 234 | 1040 | 2208 | 2273 |
| Min | 5753 | 0 | 594 | 1265 | 1742 |
| Max | 5762 | 468 | 1485 | 2523 | 3209 |

Table 25: (b) “lu” simulation on MOESI_CMP_token system for 64*64 matrix

# Chapter 5

# 5 Conclusion and Future Work

## 5.1 Conclusion:

We have studied coherence techniques in multi-core processors. Snoopy coherence method simulation result is analyzed with respect to varying block size and cache size. As observed snoop traffic plays critical role in performance of snoopy coherence technique. Multiple directory coherence protocols available in Gem5 are compared through simulation using SPLASH-2 benchmark. Protocols are compared on the basis of their architecture (1-Level cache, shared L2 cache, private L2 and shared L3 cache) and network traffic. Overall, the cache coherence techniques and protocols are explored using Gem5 simulator.

## 5.2 Future Work:

- There is further scope for better coherence protocol as each existing protocols has certain limitations. Good bandwidth utilization, less network traffic, need to be maintained. Off-chip communication should be reduced as to reduce long latency for off-chip memory access.
- MESIF protocol is not present in Gem5 suite. It would be interesting to add three level cache hierarchy with implementation of MESIF protocol in Gem5.
- Gem5 has implementation of MI, MESI, MOESI protocol with respect to directory coherence technique. Snoopy variant of above protocols will be more helpful to get clear picture of coherence protocols.
- A more detailed simulation "status" file should be generated in Gem5. The output parameters mentioned in "status" file need to be well organized.

# Bibliography

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *ACM SIGARCH Computer Architecture News*, volume 16, pages 280–298. IEEE Computer Society Press, 1988.

[2] Carnegie Mellon University. Available at http://www.ece.cmu.edu.

[3] Interconnection Network. Available at http://www.m5sim.org/Interconnection_Network, December 2013.

[4] MESI Two Level. Available at http://www.m5sim.org/MESI_Two_Level, March 2014.

[5] Nick Barrow-Williams, Christian Fensch, and Simon Moore. A communication characterisation of splash-2 and parsec. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 86–97. IEEE, 2009.

[6] Tuan Bui, Brian Greskamp, I-Ju Liao, and Mike Tucknott. Measuring and characterizing cache coherence traffic.

[7] Computer Architecture Lecture 31: Multiprocessor Correctness and Cache Coherence. Available at http://www.ece.cmu.edu.

[8] MOESI CMP directory. Available at http://www.m5sim.org/MOESI_CMP_directory, July 2013.

[9] MI example. Available at http://www.m5sim.org/MI_example, july 2013.

[10] Vivio 5.1 A Tool For Creating Interactive Reversible E-Learning Animations for the WWW. Available at https://www.cs.tcd.ie/Jeremy.Jones/vivio/vivio.htm, March 2010.

[11] José M Garcıa and Manuel E Acacio. Validating a token coherence protocol for scientific workloads.

[12] The gem5 Simulator System.A modular platform for computer system architecture research. Available at http://www.gem5.org, Feb 2013.

[13] James R Goodman and Herbert HJ Hum. Forward state for use in cache coherency in a multiprocessor system, July 26 2005. US Patent 6,922,756.

[14] MOESI hammer. Available at http://www.m5sim.org/MOESI_hammer, July 2013.

[15] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2012.

[16] Rodney Van Meter. http://web.sfc.keio.ac.jp/ rdv/keio/sfc/teaching/architecture/computer-architecture-2013/lec10 dsm.html, December 2013.

[17] Jeremy Jones. Vivio-a system for creating interactive reversible e-learning animations for the www. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 131–133. IEEE, 2004.

[18] Alberto Alcón Laguéns, Sergio Barrachina Mir, and Enrique S Quintana Ortí. An interactive animation for learning how cache coherence protocols work. *INTED2011 Proceedings*, pages 6128–6132, 2011.

[19] David J Lilja. Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons. *ACM Computing Surveys (CSUR)*, 25(3):303–338, 1993.

[20] Milo MK Martin, Mark D Hill, and David A Wood. Token coherence: decoupling performance and correctness. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 182–193. IEEE, 2003.

[21] Alberto Ros, Manuel E Acacio, and José M García. A direct coherence protocol for many-core chip multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 21(12):1779–1792, 2010.

[22] Bryan Schauer. Multicore processors–a necessity. *ProQuest discovery guides*, pages 1–14, 2008.

[23] Michael E Thomadakis. The architecture of the nehalem processor and nehalem-ep smp platforms. *Resource*, 3:2, 2011.

[24] MOESI CMP token. Available at http://www.m5sim.org/MOESI_CMP_token, July 2013.

[25] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36. ACM, 1995.