# Cache Coherence in Multi Processors Architecture

**A Dissertation submitted in partial fulfillment of the requirement for the**

**Award of degree of**

**MASTER OF TECHNOLOGY**

**IN**

**COMPUTER TECHNOLOGIES AND APPLICATIONS**

By

**POOJA ARORA**
**College Roll No. - 25/CTA/2010**

**Under the guidance of**

**Mr. MANOJ KUMAR**

**Associate Professor**
**Delhi Technological University**


**Department of Computer Engineering**

**Delhi Technological University**

**2011-2012**

# CERTIFICATE


**DELHI TECHNOLOGICAL UNIVERSITY**

BAWANA ROAD, DELHI – 110042

Date: 25/06/2012

This is to certify that dissertation entitled "**Cache Coherence in Multi Processors Architecture**" has been completed by Pooja Arora in partial fulfillment of the requirement of major project of **Master of Technology** in **Computer Technologies and Applications**.

This is a record of his work carried out by him under my supervision and support during the academic session 2011 -2012.

**(Mr. Manoj Kumar)**

**PROJECT GUIDE**

**ASSOCIATE PROFESSOR**

**(Dept. of Computer Engineering)**

**DELHI TECHNOLOGICAL UNIVERSITY**

**ACKNOWLEDGEMENT**

First of all, let me thank the almighty god and my parents who are the most graceful and merciful for their blessing that contributed to the successful completion of this project.

I feel privileged to offer sincere thanks and deep sense of gratitude to **Mr. MANOJ KUMAR**, project guide for expressing her confidence in me by letting me work on a project of this magnitude and using the latest technologies and providing their support, help & encouragement in implementing this project.

I would like to take this opportunity to express the profound sense of gratitude and respect to all those who helped us throughout the duration of this project. DELHI Technological University, in particular has been the source of inspiration, I acknowledge the effort of those who have contributed significantly to this project.

**(POOJA ARORA)**

**Master of Technology**

**(Computer Technologies and Applications)**

**Dept. of Computer Engineering**

**DELHI TECHNOLOGICAL UNIVERSITY**

# ABSTRACT

Appropriate solution to illustrious Cache Coherence Problem in shared memory multiprocessors system is one of the crucial issue for improving system performance and scalability. In this paper we have surveyed various cache coherence mechanisms in shared memory multiprocessor. Various hardware based and software based protocol have been investigated in depth including recent protocols. We have concluded that hardware based cache coherence protocol are better than software based protocol according to presently available protocols, but hardware based protocol have added the cost to implement them. As software based cache coherence protocol are more economical therefore more devotion is needed for software based protocol as they show great promise for future work.

After thoroughly studying about MESI protocol and MARSSx86 (Micro Architectural and System Simulator) simulator, which is an open source therefore its code is available without a hitch. In this project we have enhanced the performance of the system. While level 2 cache as shared we have made existing invalid to invalid transition zero at the Level 1 Data Cache at user level with dual cores and reduced this transition at the great extent when it comes to the FERRET, SWAPTIONS and CANNEAL programs of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark. The experiment results have proved that with dual cores we have increased cycles per second for above mentioned programs of PARSEC benchmark and at quad cores we have increased commits per second. When it comes to octet cores we have enhanced the commits per second for FERRET, cycles per second for SWAPTIONS AND CANNEAL program of PARSEC benchmark.

While keeping level 2 cache as private we have also enhanced the system performance in terms of cycle per second and commits per second by modifying the existing Invalid to Invalid (II) in MESI protocol's code of the MARSSx86 simulator. In fact by doing so we have successfully made invalid to invalid transition zero for the programs of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark for dual cores. Experiments have shown that for quad cores configuration we have reduced invalid to invalid transition significantly by 99% on an average. When we tested for octet cores configuration invalid to invalid transition is decreased by 99% with CANNEAL, 99% with FERRET and 70% with SWAPTIONS. As shown in experimental results we are actually depreciating the bus traffic and improving the system performance.

# TABLE OF CONTENTS

# Research Publication

- My research paper "A SURVEY ON CACHE COHERENCE PROTOCOLS WITH SHARED MEMORY MULTIPROCESSORS" is published at international conference ICACSSE' 2012.

- My research paper "A SURVEY ON CACHE COHERENCE PROTOCOLS WITH SHARED MEMORY MULTIPROCESSORS" is published at international journal IJCSIA'2012 (April Issue).

- My research paper "IMPROVING THE PERFORMANCE OF MESI PROTOCOL WITH LEVEL 2 CAHCE AS SHARED FOR MULTICORE PROCESSOR ON MARSSx86 SIMULATOR" is published at international ICCSE'2012.

- My one research paper is accepted at international journal IJCSI Volume 9 Issue 4 IMPROVING PERFORMANCE OF MESI CACHE COHERENCE PROTOCOL FOR MULTI-CORE PROCESSOR ON MARSSx86 SIMULATOR.

# List of Figures

# List of Tables

# Chapter 1: Introduction

The demand of multiprocessors is growing continuously in recent years and commercial machines with tens of processors are readily available today. In 2000, the sales of shared-memory systems with more than eight processors passed $16 billion [1]. This has been driven by the continuing need for computational power beyond what state-of-the-art uniprocessor systems can provide.

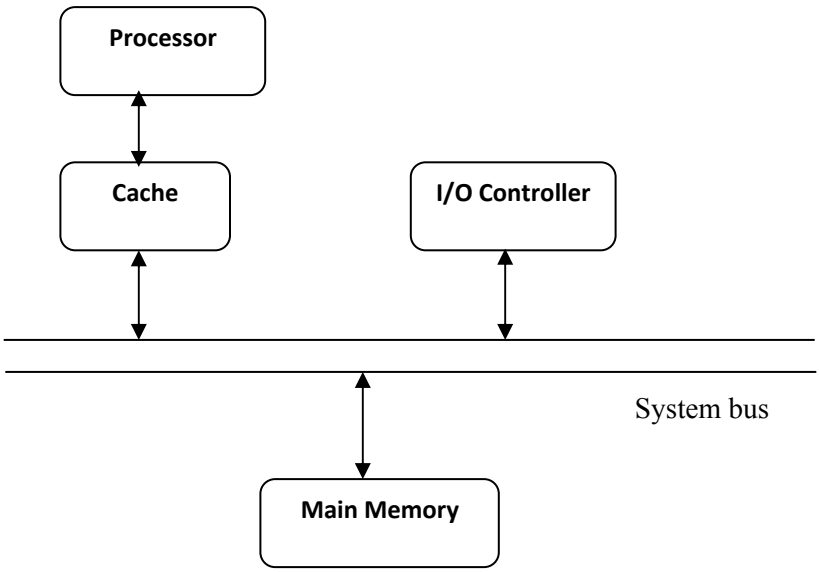



**Figure 1.1: Typical Uniprocessor Cache Configuration**

Multiprocessors architecture varies depending on the size of the machine and differs from vendor to vendor. Shared-memory architectures have become dominant in small and medium-sized machines that have up to 64 processors. They provide a single view of memory, which is shared among multiple processors, and a shared memory model for programming, where communication is achieved through accesses to the same memory location. The success of this model is due to the ease of transition it provides from uniprocessor to multiprocessors. The

programming model is similar to uniprocessor and it allows for the increased parallelization of sequential code, while achieving a very good performance.

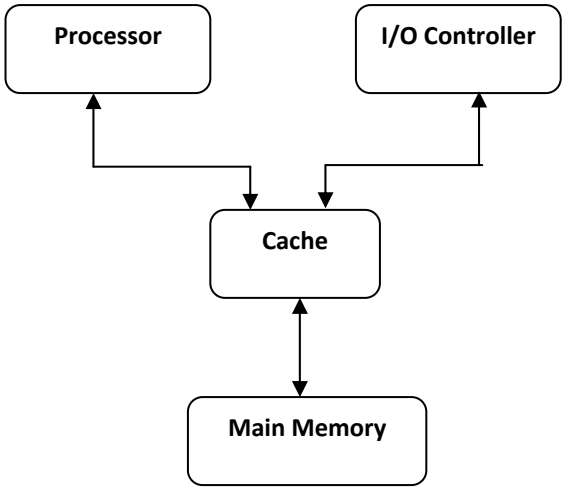



**Figure 1.2: Alternate Cache Configuration**

Shared-memory DSM (Distributed Shared Memory) machines require a coherence protocol to manage the replication of data and to ensure that a parallel program sees a consistent view of memory [2][3][4][5][6]. In general, coherence protocols allow at most a single 2 processor to modify a shared location, either invalidating outstanding copies or updating copies with the new value. A protocol determines, to a large extent, the performance of a shared-memory program since communication occurs through loads and stores to shared data. But, applications have very different patterns of communication, and no single, general-purpose protocol has proven well suited to all programs. This has prompted interest in systems that enable users to select from a set of coherence protocols [7][8] and, more recently, in systems in which a protocol is implemented in flexible software instead of being forever encoded in hardware [9][10].

To achieve high performance, the shared view of memory is implemented in hardware. The predominant architecture for small systems is based on a bus. At about 32 processors, this architecture reaches its limits [11]. For larger systems, other types of interconnection networks,

often hierarchical, are used and the memory is distributed throughout the machine. This type of architecture is referred to as a distributed shared-memory multiprocessor.

As in uniprocessors caching is used to achieve good performance, in multiprocessors it reduces the latency of accesses by bringing the data closer to the processor and it also reduces the communication traffic and bandwidth requirements in the network by satisfying requests without having to access the network. Processors typically have primary and secondary caches and the multiprocessors itself may have higher-level caches as well. The importance of caching continues to increase as systems become large and have multiple levels of hierarchy. Achieving the shared memory model in the presence of caches requires special mechanisms to maintain a coherent view of memory. These mechanisms enforce a cache coherence protocol and are usually implemented in hardware for performance. The choice of coherence protocol and its implementation play an important role in the performance of a multiprocessor system.

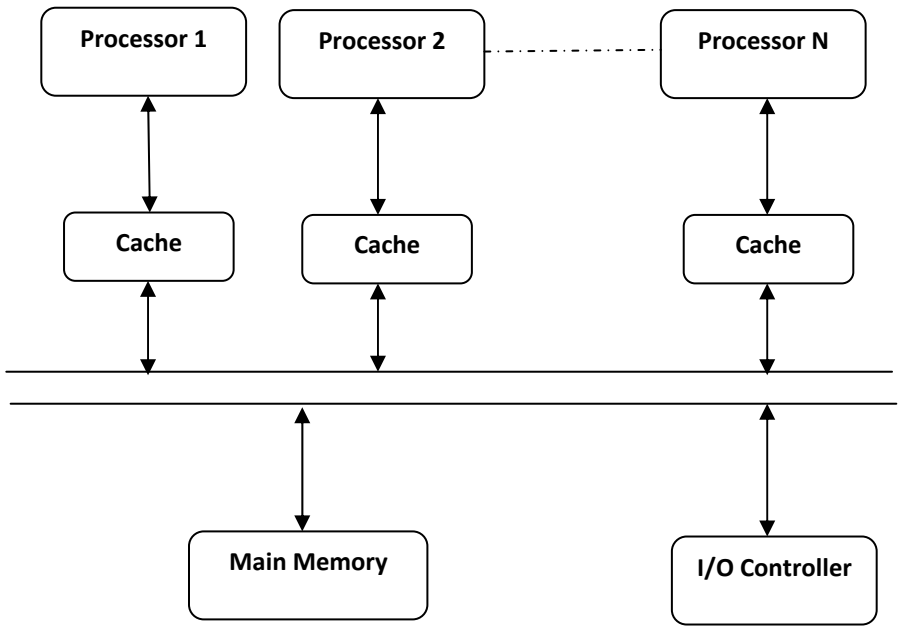



**Figure 1.3: Typical Shared Bus Multiprocessors Architecture**

## 1.1 Motivation

Much of the computer systems research over the last decade has focused on systems whose main goals are high performance and scalability to hundreds of processors. The commercial success of such multiprocessors in industry is mild. Successful multiprocessors, achieving wide-spread use, have been relatively small-scale systems. They exhibit good performance, cost-effectiveness and usability. The architectures of these systems are usually based on a bus and are built with commodity components to keep costs low. As the market continues to grow, medium-scale machines with tens of processors are emerging in a reasonable price range. The best choice of design alternatives for a multiprocessor that can scale to the medium range, up to 64 processors, is not clear still.

The primary motivation underlying parallel computing is simple: Users can obtain higher performance by distributing a computation across a set of processors and running those portions concurrently. Unfortunately, as many have discovered, programming parallel computers can be requires lots of effort than programming sequential computers. The task is easier if a parallel system supports a shared address space, since this abstraction allows processors to share a common pool of memory and frees a programmer from the concern of correctness of the data layout and movement. Distributed Shared Memory (DSM) computers, which partition the physical memory among a collection of workstation like computing nodes, are emerging as a popular way to implement parallel computers because they assure scalability and high performance.

The key for any multiprocessors system is the interconnection network. It directly affects cost, performance and usability. For medium to large-scale distributed shared memory (DSM) multiprocessors, the long latency of accesses to remote data is an issue which is becoming larger as processor speeds continue to increase faster than the speed of memory and interconnection networks. In addition, the advances in processor technology and increases in system sizes also increase the communication demands. The importance of the interconnection network has been recognized by both academia and industry.

A variety of cache coherence protocols exist and differ mainly in the scope of the sites that are updated by a write operation. These protocols can be complex and their impact on the performance of a multiprocessor system is complex to assess. The performance of a system is directly related to the latency associated with processor accesses. The latency of an access often depends on congestion in the system, which is directly related to the amount of communication traffic. Analyzing the processor data sharing behavior and determining its effect on the cache coherence communication costs is the first step in understanding the overall performance.

## 1.2 Research Objective

In this thesis we have targeted the MESI protocol existing in MARSSx86 (Micro Architectural and System Simulator) simulator. After thoroughly studying the MESI (Modified, Exclusive, Shared and Invalid) protocol and the code of MARSSx86 (Micro Architectural and System Simulator) simulator, it was found that in the existing code of MESI (Modified, Exclusive, Shared and Invalid) protocol invalid to invalid transition still exists there. We thought if it can be modified or eliminated then it should increase the system performance by making this transition zero or reducing it at very high extent and in terms of cycles per second and commits per second

also. We are actually trying to overcome this deficiency of MESI (Modified, Exclusive, Shared and Invalid) protocol or optimizing it.

## 1.3 Related Work:

We have thoroughly studied the existing code of MARSS (Micro Architectural and System Simulator) simulator and noticed that invalid to invalid transitions exist there. After doing lots of research on this transition we concluded that if this transition can be eliminated or can be reduced then it can enhance the system performance and also it can change cycles per second and commits per second. Then we initiated with the standard results, while taking these we kept level-2 cache as both private and shared.

We used PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark for getting all the results. We keep on differing number of cores viz. 2 (dual), 4 (quad), and 8 (octet) for getting standard results and results with modified code. After getting the standard results of our experiment on CANNEAL, FERRET and SWAPTIONS (programs of PARSEC benchmark) we proceeded with the actual task of modifying the code of MESI protocol. After extensive study of MARSSx86 simulator's code which is available without a hitch as open source, we have transformed the code of the file which contains the nitty-gritty of MESI protocol while executing with MARSSx86 simulator. In this file invalid to invalid transitions are present. We have modified the code precisely at this segment. Then again compiled and run this code with CANNEAL, FERRET and SWAPTIONS (programs of PARSEC benchmark) with the variation of number of cores i.e. 2, 4 and 8. After getting experiment results in the form of stats files (results of experiments comes in the form of stats

file) we have thoroughly compared them with the standard result's stats files. We have noticed that by keeping number of cores 2 i.e. dual core we have made invalid to invalid transition as zero successfully for all 3 programs of PARSEC benchmark viz. CANNEAL, FERRET and SWAPTIONS.

## 1.4 Organization of Thesis

In this chapter, we have highlighted the problems faced by users in the cache coherence with shared memory multiprocessors and multiple cores which serves as the motivation for the work reported in this thesis. Furthermore we have also outlined the specific objective of our research and related research work that has occurred in the past.

Chapter 2 provides a brief overview of the basics and concepts of coherence in cache memories and policies of cache replacement policies. It is also representing the write policies of cache viz. write through and write back policies.

Chapter 3 introduces the software and hardware based cache coherence strategies. Its giving the detail overview of all the existing software and hardware based cache coherence protocols viz. MSI, MESI, MOSI, MOESI, DRAGON protocol, Snoopy protocol, Directory based protocol and Hybrid protocol.

Chapter 4 is representing the overview of the MARSS (Micro Architectural and System Simulator) simulator and the program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark used for experimental results.

Chapter 5 presents the proposed idea on which we have worked through this project. Chapter 6 presents the performance study conducted on the proposed idea. Each conducted experiment is

discussed and detailed comments on the results are given. Finally, Chapter 8 concludes the thesis and gives some suggestions for future work.

# Chapter 2: Concepts and Background

This chapter describes the basics and terminology for understanding caches memories, interconnection networks, and the underlying problem of cache coherence in multiprocessors systems. Although this chapter reviews some concepts of cache coherence and interconnection networks, it is not an introduction to them. Rather, it is intended to give insight on the different concepts related to cache memories; in further chapters we can have insight to cache coherence protocols. We refer the reader to the established textbooks on this topic for further background and introductory material (e.g., [12][13][14][15]).

## 2.1 Basis of Cache Memories

No one memory technology can supply all the memory needs of a computer since fast memories are usually low capacity memories (low bit density). As a consequence, they are expensive: cost per bit increases as access time decreases.

Consequently, several memory types with very different physical properties placed at different levels of the memory hierarchy have to be used in typical computer systems. Main memory is a large (but slow) memory implemented with DRAM technology. To reduce the speed disparity between CPU and main memory, one or more intermediate small-sized memories called *caches* are used. The term cache refers to a fast intermediate memory within a larger memory system [16][17]. Caches, which might be implemented with SRAM technology, directly address the Von Neumann bottleneck by providing the CPU with fast access to memory.

Caches store copies of items located in main memory. Memory words are stored in a *cache data memory* and are grouped into small pages called *cache blocks* or *lines*. The contents of the cache's data memory are thus copies of a set of main memory blocks. Each cache block is marked with its block address, referred to as a tag, so the cache knows to what part of the

memory space the block belongs. The collection of tag addresses currently assigned to the cache is stored in the *cache tag memory*. Note that, for a cache to improve the performance of a computer, the time required to check tag addresses and access the cache's data memory must be much lower than the time required to access main memory.

When the CPU issues a memory address, the cache compares it to the contents of its tag memory. If a match (*hit*) occurs, the memory access is completed by the cache; otherwise (*miss*), a block that includes the addressed item is retrieved from main memory and placed into the cache. *Temporal locality* tells us that we are likely to need this word again in the near future, so it is useful to place it in the cache where it can be accessed quickly. *Spatial locality* tells us that there is a high probability that the other data in the block will be needed soon. Hence, because of locality principle and the higher speed of smaller memories, a memory hierarchy can substantially improve performance. A basic measure of this performance is the hit ratio, which is the fraction of all memory references that are satisfied by cache.

### 2.1.1 Block Placement Policy

When a block is retrieved from main memory, a *block placement* policy is used to know where the newly entered block can be placed into the cache. This policy influences when a tag address is presented to the cache, since it must be quickly compared to the stored tags to determine whether a matching occurs. Depending on the restrictions on where a block can be placed, we can categorize placement policy in three categories of cache organization given as below:

• If each memory block has only one place where it can be allocated in the cache, the cache is said to be *direct mapped*. In this case, the cache is divided into sets, each of which stores a block. With direct mapping, each block in main memory is mapped into one specific block of cache.

The main drawback of this organization is that the cache's hit ratio drops sharply if two or more frequently used blocks map onto the same region in the cache (known as collision), whereas the main advantage is its simplicity.

• If a block can be placed anywhere in the cache, the cache is said to be *fully associative*. Associative memories are also commonly known as *content-addressable memories* (CAMs). To implement fast tag comparison, the input tag can be compared simultaneously to all tags in the cache tag memory. The main disadvantage of this kind of memory is that they are expensive and complex.

• If a block can be placed in a restricted set of places in the cache, the cache is *set associative*. A set is a group of blocks in the cache. A block in main memory is first mapped onto a set, and then the block can be placed anywhere within that set. If there are m blocks in a set, the cache placement is called m-*way set associative*.

This approach reaches a trade-off between the advantages and disadvantages of the two previous proposals. Thus, it is considered a reasonable compromise between the complex hardware needed for fully associative caches (which requires parallel searches of all tags), and the simple direct-mapped scheme. The main disadvantage is similar to that in direct mapped caches, since collisions may occur.

### 2.1.2 Replacement Policy

When a miss occurs, a cache block must be selected to be replaced with the block retrieved from main memory. The main advantage of direct mapped policy is that hardware decisions are simplified since a replacement policy is not required: each block has only one place to be placed and only that block can be replaced. With fully associative or set-associative placement, there are

many blocks to choose from on a miss. The most employed strategies for selecting the block to replace are:

• ***Random.*** The candidate block to replace is randomly selected.

• ***Least-recently used (LRU).*** Relying on the past to predict the future, the block replaced is the one that has been unused for the longest time.

• ***First in, first out (FIFO).*** Because LRU can be complicated to calculate, this approximates LRU by determining the oldest block rather than the least-recently used one.

### 2.1.3 Write Policy

Another important aspect of caches is the write policy. There exist two different strategies when a write is carried out: write-through and write-back. In write-through, the information is written to both the block in cache and to the block in main memory. This policy is easy to implement, and it assures that main memory will never have stale information. In write-back, the information is written only to the block in cache. This modified cache block is written to main memory only when it is replaced because of any requirement. This technique has the disadvantage of temporal inconsistency, that is, cache and main memory can have different data associated with the same physical address. In addition, the write-back technique complicates recovery from system failures. On the other hand, write-through results in more write cycles to main memory than write-back does.

To reduce the frequency of writing-back blocks on replacements, a feature called the *dirty bit* is commonly used. This status bit indicates whether the block is *dirty* (modified while in cache) or *clean* (not modified). If it is clean, the block is not written back on a miss, since identical information to the cache is found in main memory.

### 2.1.4 Structure

Table 2.1 illustrates some of the diversity of commercial cache types. As clock speeds separated from main memory speeds, fast and small cache memories began to be included to boost performance. Thus, early computers employed a single, multichip cache that occupied one level of the hierarchy between the CPU and main memory. These caches were external to the processor and located on the motherboard (some versions of the *386* processor could support up to 64 KB of external cache). Later, due to the feasibility of including part of the real memory space on a microprocessor chip and the growth in the size (but not in the speed) of main memory, more cache levels were introduced, which addressed the increase of the miss penalty. A Level 1 (L1) cache is an efficient way to implement an on-die memory. It was named like that to differentiate it from the Level 2 (L2) cache, which was still located on the motherboard (off-die). The L2 cache is slower than L1 cache, but it is much larger. In general, caches of levels close to the CPU are smaller, but faster than caches of higher levels. Hence, with the appearance of the *486* processors (and later in the *Pentium MMX*), an 8 KB cache began to be integrated directly into the CPU die. Later, the introduction of SDRAM to implement main memory and the growing difference between the bus speed and the CPU clock speed caused on-motherboard cache to be only slightly faster than main memory, which forced a new evolution. Thus, some processors such as *Pentium Pro*, *Pentium II*, and the first *Pentiums III* incorporated the secondary cache into the same cartridge as the CPU, but out of the die.

| Model | L1 cache | L2 cache | L3 cache |
|---|---|---|---|
| Intel 386 | off-die 64 KB | (none) | (none) |
| Intel 486 | on-die 8 KB | (none) | (none) |
| Pentium MMX | split on-die 16 KB | (none) | (none) |
| Pentium Pro | split on-die 8 KB | cartridge 1 MB | (none) |
| Itanium 2 | split on-die 16 KB | split on-die 1 MB | on-die 12 MB |
| Xeon MP | split on-die 8 KB | on-die 2 MB | on-die 16 MB |
| IBM Power 4 | split on-die 96 KB | on-die 1 MB | off-die 256 MB |
| AMD Phenom | split on-die 64 KB | on-die 512 KB | on-die 2 MB |

**Table 2.1: Cache features in actual computers**

The desirability of additional levels increases with the size of main memory. As main memory size increases further, the latency difference between main memory and the fastest cache becomes larger. This makes even more cache levels to be desirable (for example, a third level of cache). This level can be implemented on a separated chip from the CPU (the *IBM Power 4* series support up to 256 MB L3 cache off-chip) or incorporated in the same chip (*Itanium 2* incorporated a 12 MB L3 cache on-die, the *AMD Phenom* series of chips carries a 2 MB on-die L3 cache, and the *Intel Xeon MP* features 16 MB on-die L3 cache).

Multi-level caches can be classified in different types. A cache is said to be *strictly inclusive* when all data in L1 cache are also in L2 cache. Other processors (like the *AMD Athlon*) have *exclusive* caches, that is, a datum is either in L1 cache or in L2 cache, never in both.

Caches are also distinguished by the kind of information they store. An *instruction* or *I-cache* stores instructions only, while a *data* or *D-cache* stores data only. Separating the stored data in this way recognizes the different access behavior patterns of instructions and data. A cache that stores both instructions and data is referred to as *unified* (such as in the *PA-7100 LC* processors [18]). On the other hand, a *split cache* consists of two associated but largely independent units:

an I-cache and a D-cache. While a unified cache is simpler, a split cache makes it possible to access programs and data concurrently.

## 2.2 Cache Coherence

Although the microprocessor performance has been improving at a rate of about 50% per year, it may be increasingly difficult that a single processor becomes fast enough to satisfy the applications demands for ever greater performance.

An attractive solution can be the parallel machines, since they are built from multiple conventional, small, inexpensive, low-power, massproduced processors.

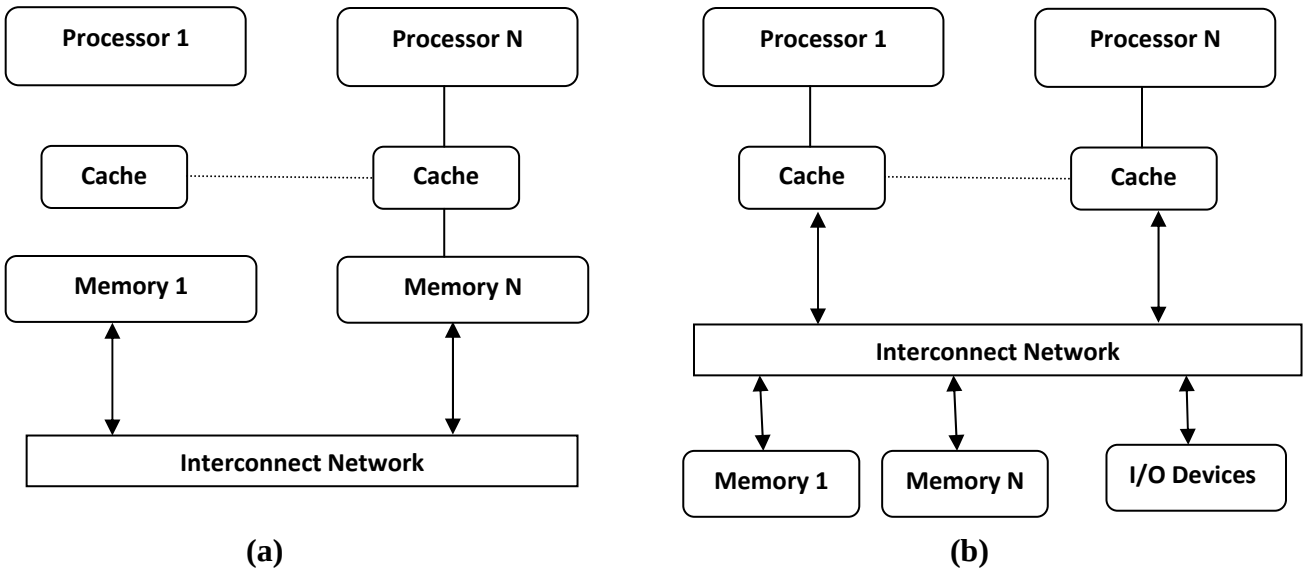



**Figure 2.1(a) Multicomputer and (b) UMA Architecture**

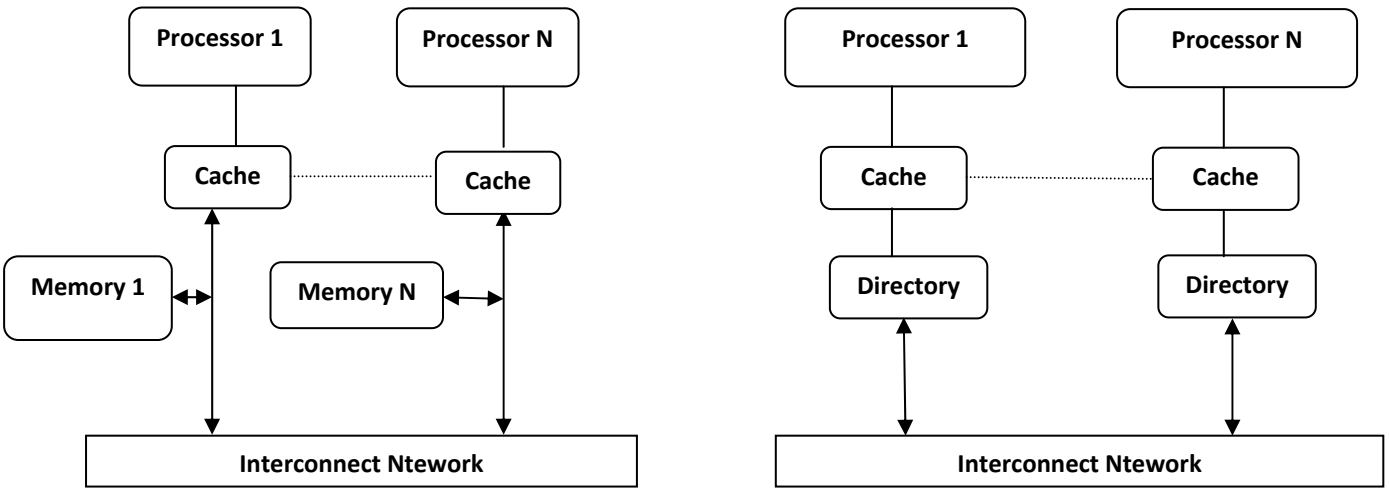
Parallel machines are based on the MIMD architecture (*Multiple Instruction stream, Multiple* Data stream) and are usually classified in two different types: multicomputers and multiprocessors.

In multicomputer systems, each processor has its own local memory. Therefore, the global memory of the system is physically distributed among all the processors as shown in Figure 2.1(a). Each processor is tightly coupled to its memory, which, besides being physically separate,

is logically private from the memories of other processors. A global memory address does not exist; rather, each processor has its own private memory address space. This kind of system is also known as message-passing multicomputer, as it is the only way several processors can communicate among themselves.

A multiprocessors is a parallel system compound of several interconnected processors which share a global physical address space that can be accessed from any processor. This kind of system is also known as shared-memory system. Depending on how the memory is shared, multiprocessors may be classified in different types:

• In UMA (*Uniform Memory Access*) systems, the access to all shared data of main memory




(c) (d)

**Figure 2.1(c) CC-NUMA and (d) COMA Architecture**

from any processor is uniform, that is, the access latency does not depend on the location of the physical address. As Figure 2.1(b) depicts, every processor has its own private cache and all the processors and memory modules attach to the same interconnect. These systems are also known as SMP (Symmetric Multiprocessing). The UMA systems that incorporate cache coherence are usually named as CC-UMA (Cache-Coherent Uniform Memory Access).

• In NUMA (Non-Uniform Memory Access) architectures, processors and memory modules are closely integrated such that the access to the local memory is faster than the access to the remote memories. Figure 2.2(a) illustrates the NUMA model, where the global memory is shared but local to each processor. This model is also known as DSM (Distributed Shared Memory). The main advantage of the NUMA architecture is that the access to the local memory is faster than that in the UMA model, although the access to a non-local memory is slower. There exists a CC-NUMA (Cache-Coherent Non-Uniform Memory Access) model with distributed shared memory and cache directories to implement coherency. Besides, there exists another alternative, NCCNUMA (Non Cache-Coherent Non-Uniform Memory Access) where data are storable in the processor's cache only if those data belong to its local memory, thereby not require maintaining coherence.

• In COMA (Cache Only Memory Access) architecture, the local main memory is managed as a hardware cache, providing replication and coherence at cache block granularity. In COMA machines, every memory block in the entire main memory has a hardware tag associated with it. There is no fixed node where space is always guaranteed to be allocated for a memory block. Rather, data dynamically move to and are replicated in the main memories that access. These main memories are organized as caches, shown in Figure 2.2(b). Some authors consider this model as a special kind of NUMA machine where the distributed local memories become caches memories. The main advantage of the COMA model is that it frees parallel software from worrying about data distribution in main memory. However, COMA machines require a lot of hardware support, they have extra memory overhead, and the required coherence protocols are complex.

Coherence defines the behavior of reads and writes to the same memory location. The coherence of caches is obtained if the following conditions are met:

1. A read made by a processor P to a location X that follows a write by the same processor P to X, with no writes of X by another processor occurring between the write and the read instructions made by P, X must always return the value written by P. This condition is related with the program order preservation, and this must be achieved even in mono-processed architectures.
2. A read made by a processor P1 to location X that follows a write by another processor P2 to X must return the written value made by P2 if no other writes to X made by any processor occur between the two accesses. This condition defines the concept of coherent view of memory. If processors can read the same old value after the write made by P2, we can say that the memory is incoherent.
3. Writes to the same location must be sequenced. In other words, if location X received two different values A and B, in this order, by any two processors, the processors can never read location X as B and then read it as A. The location X must be seen with values A and B in that order.

These conditions are defined supposing that the read and write operations are made instantaneously. However, this doesn't happen in computer hardware given memory latency and other aspects of the architecture. A write by processor P1 may not be seen by a read from processor P2 if the read is made within a very small time after the write has been made. The memory consistency model defines when a written value must be seen by a following read instruction made by the other processors.

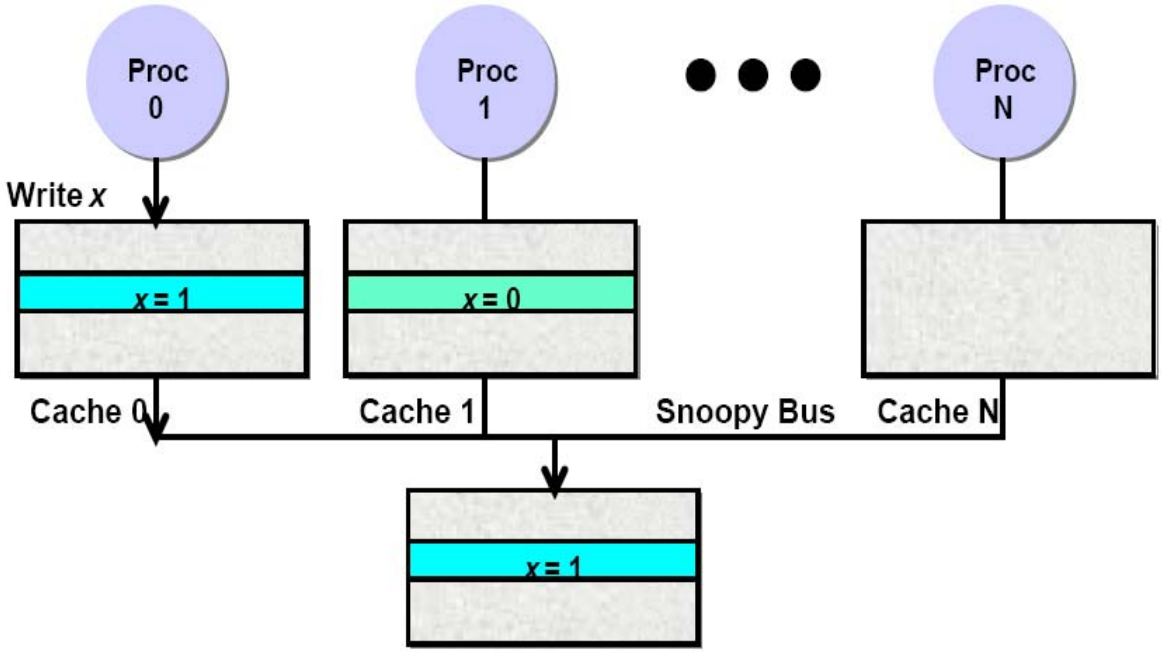



**Figure 2.2: Multiprocessors with Shared memory cache**

There exist two different options to implement cache coherence protocols. On the one hand, the protocols that invalidate cache copies (other than the writer's copy) on a write are called invalidation-based protocols. On the other hand, the protocols that update cache copies are called update-based protocols. In both cases, the next time the processor with the copy accesses the block, it will see the most recent value, thereby ensuring a coherent view of the memory system. Since invalidation-based coherence has been used in most recent systems (e.g., [19][20[21][22][23][24][25]), this dissertation only considers this kind of implementation.

## 2.3 Summary

In this chapter we have studied the basics of cache coherence in multiprocessors with shared memory architecture. We have gone through all the cache block placement and replacement strategies. We have also mentioned the write strategy of cache i.e. write through and write back strategy.

# Chapter 3: Classification of Cache Coherence Protocol

We have classified cache coherence protocol on the basis of usage of hardware. In software based solutions we do not use any hardware but in hardware based solution an additional hardware is used. As it is obvious that by using an additional hardware it will enhance the system cost. When it comes to software based solutions it is not using any additional hardware but again it is not as efficient as hardware based protocols are. The cache coherence protocols can be classified as shown below in figure 3.1.

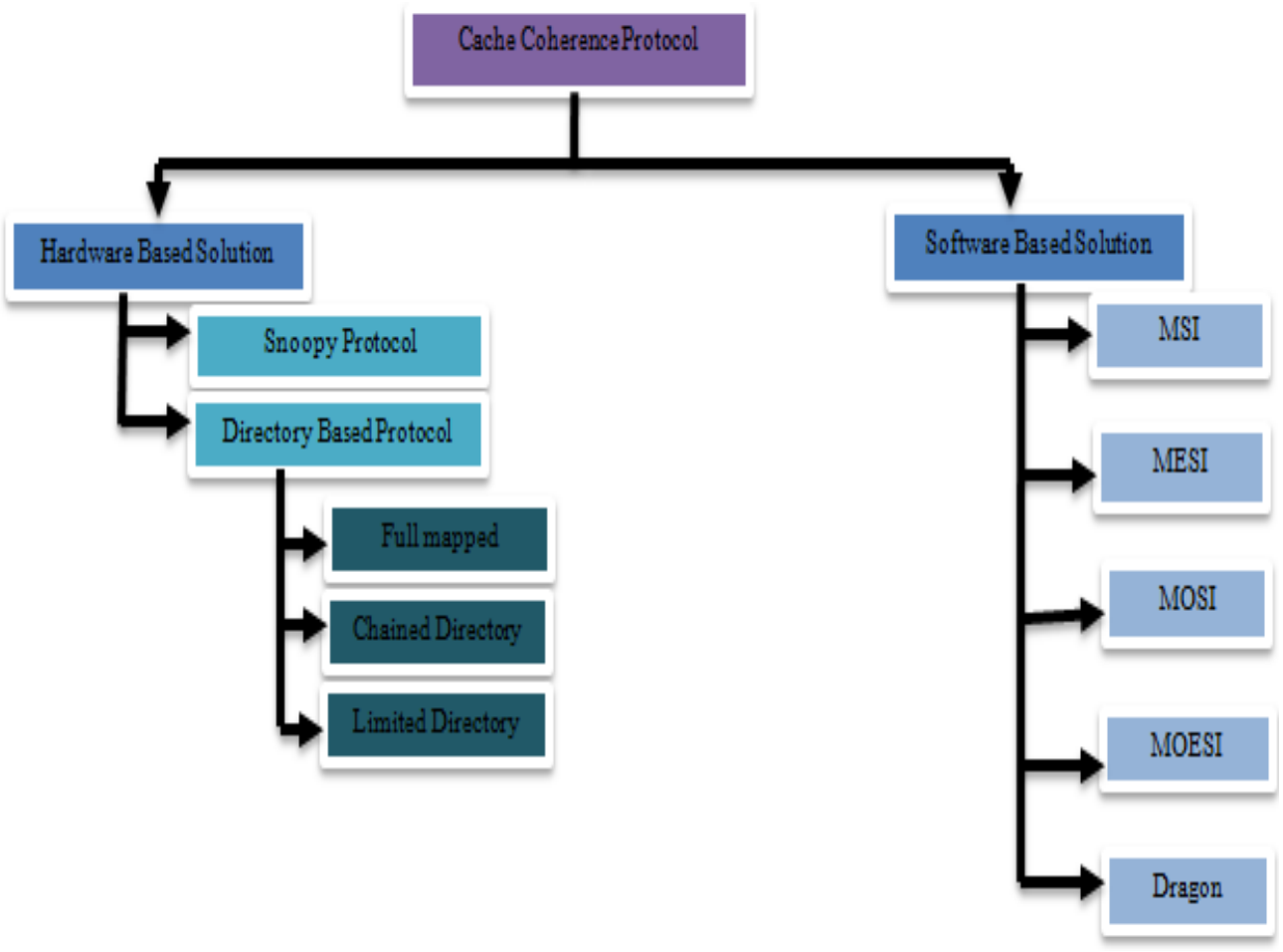



**Figure 3.1: Classification of Cache Coherence Protocol**

## 3.1 Software Based Solutions

The hardware based cache coherence protocols which will be discussed in later in this chapter are based techniques to enforce the coherence is simple and efficient, the involvement of the extra hardware can be very costly and may not be very scalable. Software based solutions generally rely on the actions of the programmer or operating system in dealing with the coherence problem. Added to this the compilers can gather good data dependence information during compilation which may simplify the coherence task. There are some methods which allow the caching of shared data and accessing them only in critical sections, in a mutually exclusive way. Decisions about coherence related actions are often made statically during the compiler analysis (which tries to detect conditions for coherence violation). There are also some dynamic methods based on the operating system actions. Optimizing compilers can then be used to reduce coherence overhead, or in combination with operating system or limited hardware support, provide the necessary coherence at a lower cost. Software approaches are generally less expensive than hardware approach, though they may require considerable hardware support. It is also claimed that they are more convenient for large, scalable multiprocessors. On the other hand software based approach has some disadvantages, especially in static schemes, where inevitable inefficiencies are incurred since the compiler analysis is unable to predict the flow of program execution accurately and conservative assumptions have to be made.

The software based approaches are classified as:

* MSI Protocol
* MESI Protocol
* MOSI Protocol

* MOESI Protocol
* DRAGON Protocol

### 3.1.1 MSI Protocol

MSI is a simple invalidation-based protocol for write-back caches. It is very similar to the protocol that was used in the Silicon Graphics 4D series multiprocessor machines [26]. The MSI protocol defines three states, modified (M), shared (S), and invalid (I), to distinguish valid blocks that are unmodified (clean) from those that are modified (dirty). Invalid means the block is not present in cache. Shared means the block is present in cache in an unmodified state, main memory is up-to-date, and zero or more other caches may also have an up-to-date (shared) copy. Modified means that only this cache has a valid copy of the block and the copy in main memory is stale. An invalid block can not be neither read nor written, a shared block can be read, but not written, and a block in the modified state can be read and written.

Before an invalid block can be read or before a shared/invalid block can be written, the processor has to order such an operation (read or write) upon the block. To this end, the MSI protocol defines two different classes of requests: write requests and read requests.

On a write miss, a write request is used to tell other caches about the impending write and to acquire an exclusive copy of the block. A cache is said to have an exclusive copy of a block if it is the only cache with a valid copy of it (main memory may or may not have a valid copy). Therefore, a write request serves to both order the write and cause the invalidation of all copies. The memory system (possibly another cache) supplies the data to the requester. Once the requester acquires the exclusive copy, the write can be performed in its cache.

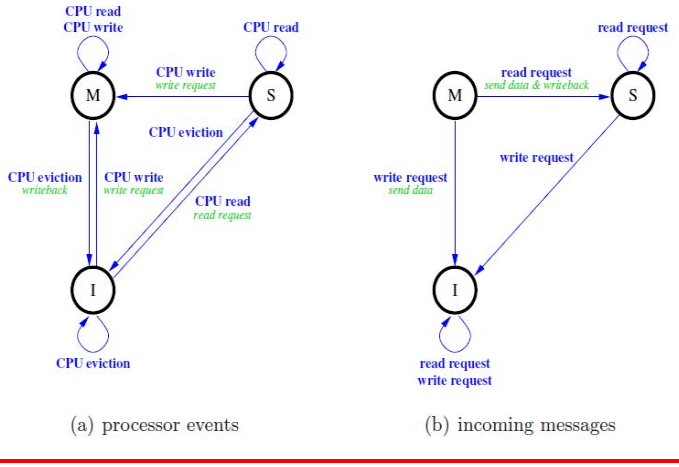



**Figure 3.2: State transition Model for MSI protocol**

On a read miss, that is, when there is no intention to modify the copy of a block, a read request is issued. The memory system (possibly another cache) supplies the data.

Figure 3.2 shows the state transition diagram that governs a block in each cache for the MSI protocol. As shown, a processor read to a block that is invalid causes the issue of a read request to service the miss. The newly loaded block transitions from invalid to shared in the requesting cache, as shown in 3.2(a). Any other caches with the block in the shared state that observe the read request take no special action, allowing main memory to respond with the data. However, if a cache has the block in the modified state and it receives a read request, then it must respond with the data, update the copy in main memory, and its copy of the block transitions to the shared state, as shown in Figure 3.2(b). It is also possible not to update the copy in main memory, leaving memory still out-of-date, but this requires more states [27].

On a write miss (writing into an invalid or shared block), a write request is issued. This request causes all other cached copies of the block to be invalidated, thereby granting the requesting cache exclusive ownership of the block.

The block in the requesting cache transitions to the modified state, and the desired bytes are then written into it. A common optimization to reduce data traffic is to introduce a new request, called upgrade request. An upgrade request obtains exclusive ownership just like a write request, by causing other copies to be invalidated, but it does not cause main memory or any other device to respond with the data for the block. Upgrade requests are useful on write misses for shared blocks.

A replacement of a block from a cache causes its eviction. This replacement causes the state machine for two blocks to change states: the one being replaced changes to invalid, and the one being brought in changes either to shared or to modified. If the block being replaced was in modified state, the block is written back to main memory. However, if the block being replaced was in shared state, a silent eviction is performed (it is not necessary to inform about the eviction).

### 3.1.2 MESI Protocol

Another aspect of the MSI model susceptible to be improved is the following: a cache with a block in modified state does not distinguish between an exclusive copy that has been modified and an unmodified exclusive copy that is only held by that cache (since any other cache does not currently have a valid copy). This situation can lead to unnecessary data traffic, as the replacement of unmodified exclusive blocks cause the blocks to be written back to main memory. Besides this problem, another concern arises when the MSI model is used in a multiprocessor running a sequential application. In this case, when a processor reads in and modifies a memory block, the MSI model generates two consecutive cache misses (even though

there are no sharers), since the first cache miss retrieves the block in shared state and the second it is necessary to convert S state to M state.

The two aforementioned situations are avoided by adding a state indicating that the block is the only (exclusive) copy but it is not modified. This new state, called exclusive (E), indicates an intermediate level of binding between shared and modified. It is exclusive, so unlike the shared state, the cache can perform a write (directly transitioning to the modified state). However, the exclusive state does not imply ownership (memory has a valid copy), so unlike the modified state, the cache does not need to reply when observing a request upon the block. Variants of this MESI protocol [28] are used in may microprocessors, including the Intel Pentium, PowerPC 601, and the MIPS R4400 used in the Silicon Graphics Challenge multiprocessors.

### 3.1.3 MOSI Protocol

The main advantage of the MSI model is its simplicity, but it has numerous drawbacks. For instance, when a cache block transitions from modified to shared, the block has to be written back to main memory, which may generate a lot of data traffic. Besides, the requests for blocks shared by two or more processors are always served by main memory, which is slow (memory-to-cache transfer). To improve these aspects, some models add a new owned state (O).

This state in a processor's cache allows read only access to the block (much like shared), but also signifies that the value in main memory is not up-to-date. In addition, a cache is said to be the owner of a block if it must supply the data upon a request for that block [27]. This permits that in some implementations of the MOSI model (such as those based on IBM NorthStar/Pulsar processors [29][30][31]) the latency of cache misses lowers since data are usually supplied by caches (cache-to-cache transfer) instead of main memory (memory-to-cache transfer).

Figure 3.3 shows the state transition diagram for the MOSI protocol. As illustrated, if a cache holds a block in modified state and it receives a read request for it, it must provide the data to the requester and its copy transitions to owned. Note that, unlike the MSI model, the block is not written back to main memory, leaving memory still out-of-date, thereby lowering the data traffic.

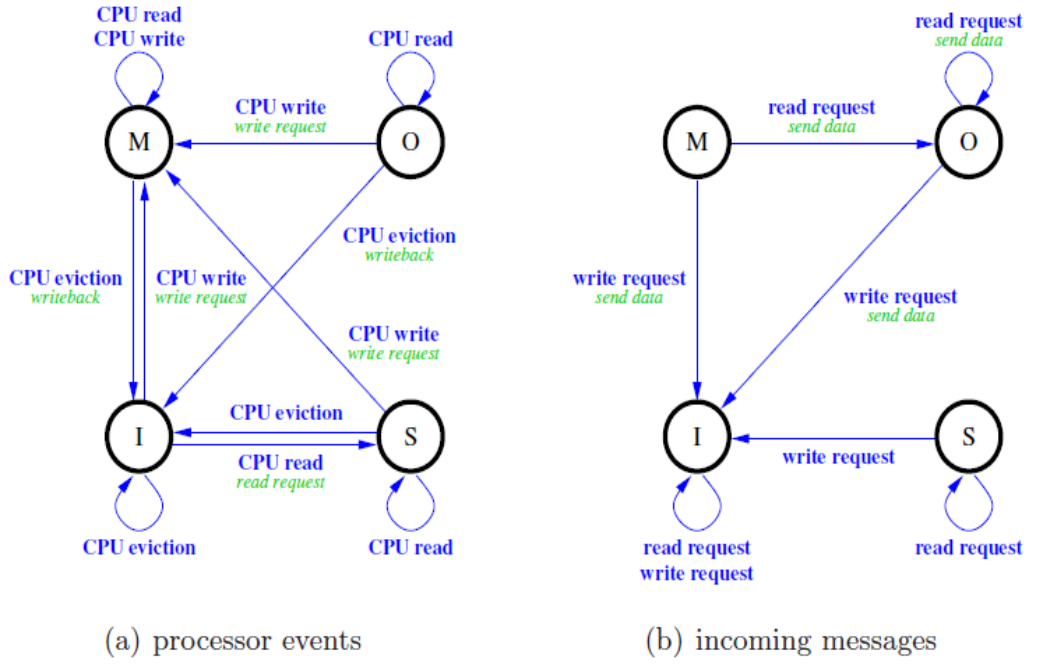



**Figure 3.3: State Transition Diagram for MOSI Protocol**

For a memory block, only one cache can have a copy of it in owned state, while the other copies of the block can be in shared state. The cache holding a block in owned state is in charge of supplying the data to all the caches that request a copy. Note that, if a read request has been observed, the owned cache remains in the same state, but if a write request has been observed, the block transitions from owned state to invalid state. Like in modified state, the replacement of a block in owned state causes the block to be written back to main memory.

### 3.1.4 MOESI Protocol

To join the major advantages of MOSI and MESI models, the MOESI protocol was proposed. Figure 3.4 shows the state transition diagram for this model. The final definition of the states is as follows. A cache has a block in modified state when it is the only valid copy of the block in the system. This copy has been modified and the copy in main memory is stale. A cache with the block in modified state can read and write that block. On a replacement, the block has to be written back to main memory. The modified state implies ownership.

Therefore the data must be supplied to both read requesters (transitioning to owned) and write requesters (transitioning to invalid). A cache has a block is in owned state when that cache and, at least, another one have a valid copy of the block. The copy in main memory may be stale, therefore, on a replacement, the block is written back to main memory. A cache with the block in owned state can only read it. Like the modified state, this state implies ownership. Therefore, it must supply the data when observing a read request (remaining in owned state) or a write request (transitioning to invalid state).

A cache has a block in exclusive state if it is the only cache with a valid copy and

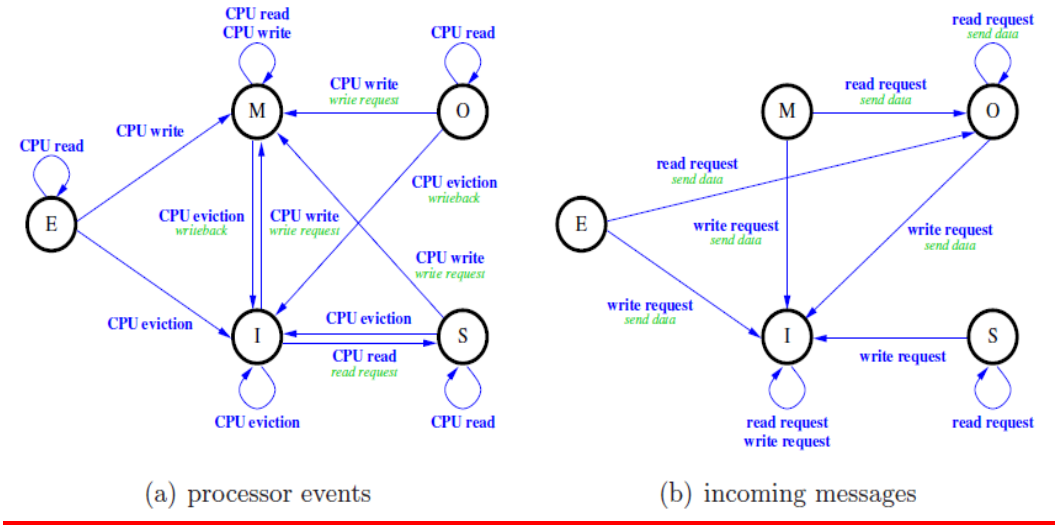



**Figure 3.4: State Transition Diagram of MOESI protocol**

its value matches with the value in main memory. On a replacement, the block does not have to be written back to main memory. A cache with a block in exclusive state can write it (making a silent transition to modified state) and read it (remaining in the exclusive state). Unlike the exclusive state defined in the MESI model, the exclusive state in the MOESI model implies ownership. Therefore, a cache with a block in that state has to serve both read requests (transitioning to owned state) and write requests (transitioning to invalid state) upon the block.

A block is in shared state when there exist several valid copies of the block throughout the system. A cache with a block in shared state can only read it. This state does not imply ownership. Therefore, it is not in charge of serving requests. A block is in invalid state when the cache does not have a valid copy of it. The issue of a request will be required to be able to access the block.

### 3.1.5 Dragon Protocol

This protocol was first proposed by researchers at Xerox PARC for their Dragon multiprocessors system. The Dragon protocol consists of four states: Exclusive-clean (or exclusive), has the same meaning as in other protocol only one cache (this cache) has a copy of the block, and it has not been modified (the main memory is up-to-date). Shared-clean, means that potentially two or more caches (including this one) have this block, and main memory may or may not be up-to-date. Shared-modified, means that potentially two or more caches have this block, main memory is not up-to-date, and it is this cache's responsibility to update the main memory at the time this block is replaced from the cache; a block may be in this state in only one cache at a time; however it is quite possible that one cache has the block in this state, while others have it in shared-clean state. Modified, state signifies exclusive ownership as before; the block is modified and present in this cache alone even main memory is stale and it is this cache's responsibility to supply the data and to update main memory on replacement.

Note that there is no explicit invalid state as in the MOESI protocols, because it is an update-based protocol. The protocol always keeps the blocks in the cache up-to-date, so it is always okay to use the data present in the cache if the tag match succeeds. However, if a block is not present in a cache at all, it can be imagined in a special invalid or not-present state.

## 3.2. Hardware Based Solutions

Implementing cache coherence protocols in hardware has been the route taken by most commercial manufacturers. Once a suitable cache coherence protocol has been defined and implemented in digital logic, it can be included at every node to manage cache operations transparently from the programmer and compiler. Although the hardware costs may be

substantial, cache coherence can be provided in hardware without compiler support and with very good performance. The increased cost is well justified by significant advantages of the hardware-based approach, these advantages are:

- Hardware schemes deal with coherence problem by *dy*namic recognition of inconsistency conditions for *shared* data entirely at run time. They promise better performance, especially for higher levels of data sharing, since the coherence overhead is generated only when actual sharing of data takes place.
- *Being* totally transparent to software, hardware protocols free the programmer and compiler from any responsibility about coherence maintenance, and impose no restrictions to any software layer.
- Various proposed hardware schemes efficiently support the full range from small to large scale multiprocessors.
- Technology advances made their cost quite acceptable, *compared* to the *system* costs. Due to aforementioned reasons, hardware cache coherence protocols are much more investigated in the literature, and also much more frequently implemented in commercial multiprocessor systems.

A variety of hardware methods were developed depending on the size of the multiprocessor. The hardware based cache coherence protocol can be classified as:

1. Snoopy Cache Coherence Protocol
2. Directory Based Cache Coherence Protocol

Snooping protocols work well for small numbers of processors, but do not scale well when the number of processors increase beyond 32. Directory-based protocols can support hundreds or thousands of processors at very good performance, but may also reach scalability barriers beyond

that point. Current commercial systems use directory-based protocols with very good performance.

### 3.2.1 Snoopy Protocol

Snooping protocols were designed based on a shared bus connecting the processors which is used to exchange coherence information. The shared bus is also the processors' path to main memory. The idea behind snooping protocols is that every write to the cache is passed through onto the bus (write-through cache) to main memory. When another processor that is caching the same data item detects the write on the bus, it can either update or invalidate its cache entry as appropriate. A processor effectively "snoops" memory references by other processors to maintain coherence.

Since an update is immediately visible to all processors, a snooping protocol generally implements strong consistency. Snooping protocols are simple, but the shared bus becomes a bottleneck for a large number of processors. Although inventive bus schemes have been proposed to increase the bus bandwidth, they often resulted in a greater memory delay. Adding more than one bus or different connection buses is limited by the fact that all processors share the relatively slow memory and bus resources. Thus, snooping protocols are limited to small-scale multiprocessors of typically less than 32 processors.

As the bus is an important commodity in these systems, various approaches were taken to reduce bus traffic. The choice between write-update and write-invalidate protocols is especially important in these systems due to the large number of messages broadcast to maintain strong consistency. Hybrid protocols between WU and WI were developed in [20] and [3] to reduce

consistency traffic. These protocols use write caches to reduce traffic in WU, and allow a cache to dynamically determine whether to invalidate or update a data item based on its access pattern.

### 3.2.2 Directory Based Protocol

Directory-based coherence protocols eliminate the need for a shared bus by maintaining information on the data stored in caches in directories. By browsing a directory, a processor can determine which other processors are sharing the data item that it wishes to access and send update or invalidate messages to these processors as required. Directory-based protocols scale better than bus-based protocols because they do not rely on a shared bus to exchange coherence information. Rather, coherence information can be sent to particular processors using point-to-point connections. A processor communicates with the directory if its actions may affect consistency. Inconsistency may occur when the processor attempts to write a shared data value. The directory maintains information about which processors cache what data blocks. Before a processor is allowed to write a data value, it must get exclusive access to the block from the directory. The directory sends messages to all other processors caching the block to invalidate it, and waits for the acknowledgements of the invalidation requests. Similarly, if the processor tries to read a block that is held exclusively by another processor P, a cache miss occurs and the directory will send a message to processor P to write back its results to memory and obtain the most current value of the block. Depending on how long the directory waits before granting block access to the requesting processor determines which consistency model is supported. If the system waits for invalidation and write-back acknowledgments before letting a processor to proceed, the system implements strong consistency. Better performance can be obtained by delaying the processor only when accessing a synchronization variable, as in weak consistency. By using directory information and a cache controller which accesses and maintains the

directory, these multiprocessor systems are no longer limited by the performance of a shared bus and/or memory.

Figure 3.5 depicts a very basic directory scheme. A system with two processors, P1 and P2, and a memory, M, is assumed. For this example, a write-back/invalidate protocol is used to maintain coherence. The directory consists of two presence bits, P1 and P2, which indicate which processors have a copy of a given cache block, and a state bit, V (valid), which indicates the status of the cache block. The memory initially has the only valid copy as shown in Figure 3.5(a). The directory information, with both presence bits set to zero and the valid bit set to one, indicates that neither processor has a copy of this cache block A. Assume that processor P1 now reads a copy of cache block A. The directory in Figure 3.5(b) indicates that P1's cache contains a copy of block A by having P1's presence bit set. Next, processor P2 wants to write A and

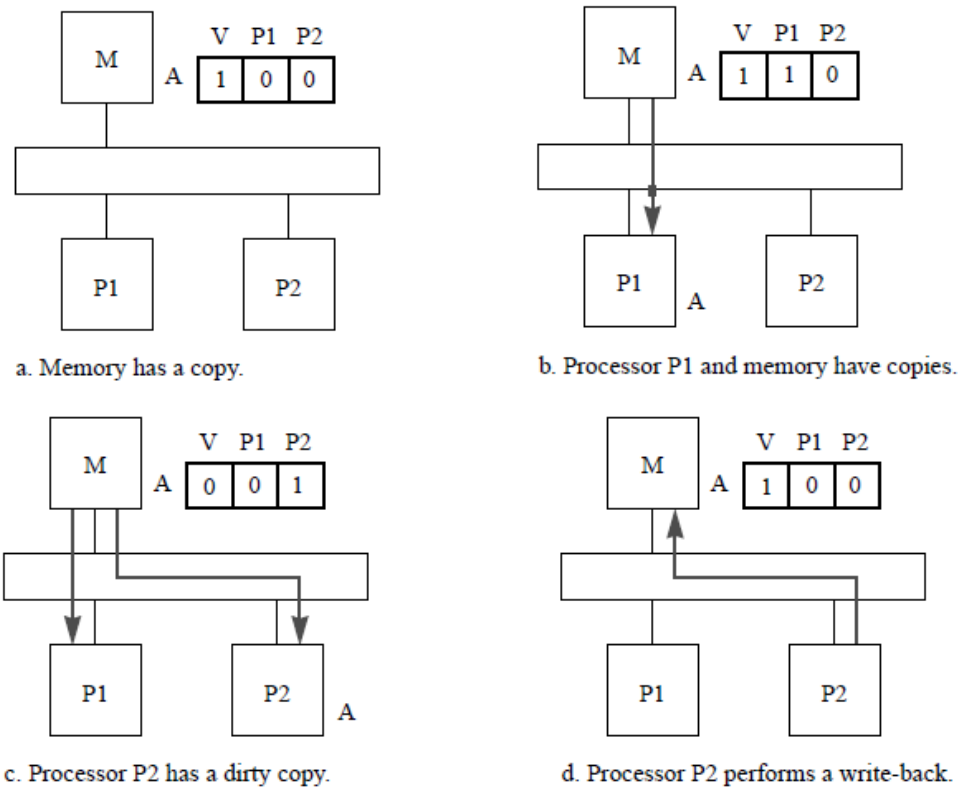



**Figure 3.5: Cache coherence with Directory based protocol**

sends a request for an exclusive copy of A to the memory. The cache coherence mechanism at the memory sends an invalidation to processor P1 followed by a copy of the cache block to P2 as shown in Figure 3.5(c). The directory reflects this change: P2 has the only (dirty) copy of the cache block which is indicated by the P2 presence bit being set to one and the valid bit being set to zero. If P2 reads another cache block B, which maps to the same location in its secondary cache, then it rejects the cache block A from its secondary cache and writes it back to the memory as shown in Figure 3.5(d). The directory updates its information indicating that the only valid copy is in the memory. Many versions of directory schemes have been proposed and many machines with hardware cache coherence have been built [32] [33] [34] [35]. When designing a

directory protocol, it is important that it perform well for typical workloads and data sharing patterns.

Now, the processors and their memories can be distributed in space, and connected with point-to-point networks which have much better scalability characteristics. Multiprocessor systems which use directory-based protocols can scale to hundreds or even thousands of processors. Many commercial multiprocessor systems implement directory-based coherence including the new SGI Origin which can have 1,024 processors in a maximal configuration.

The directory can be organized in several ways. We can categorized directory as (shown in figure 3.6):

- Full mapped Directory Protocol
- Limited Directory Protocol
- Chained Directory Protocol

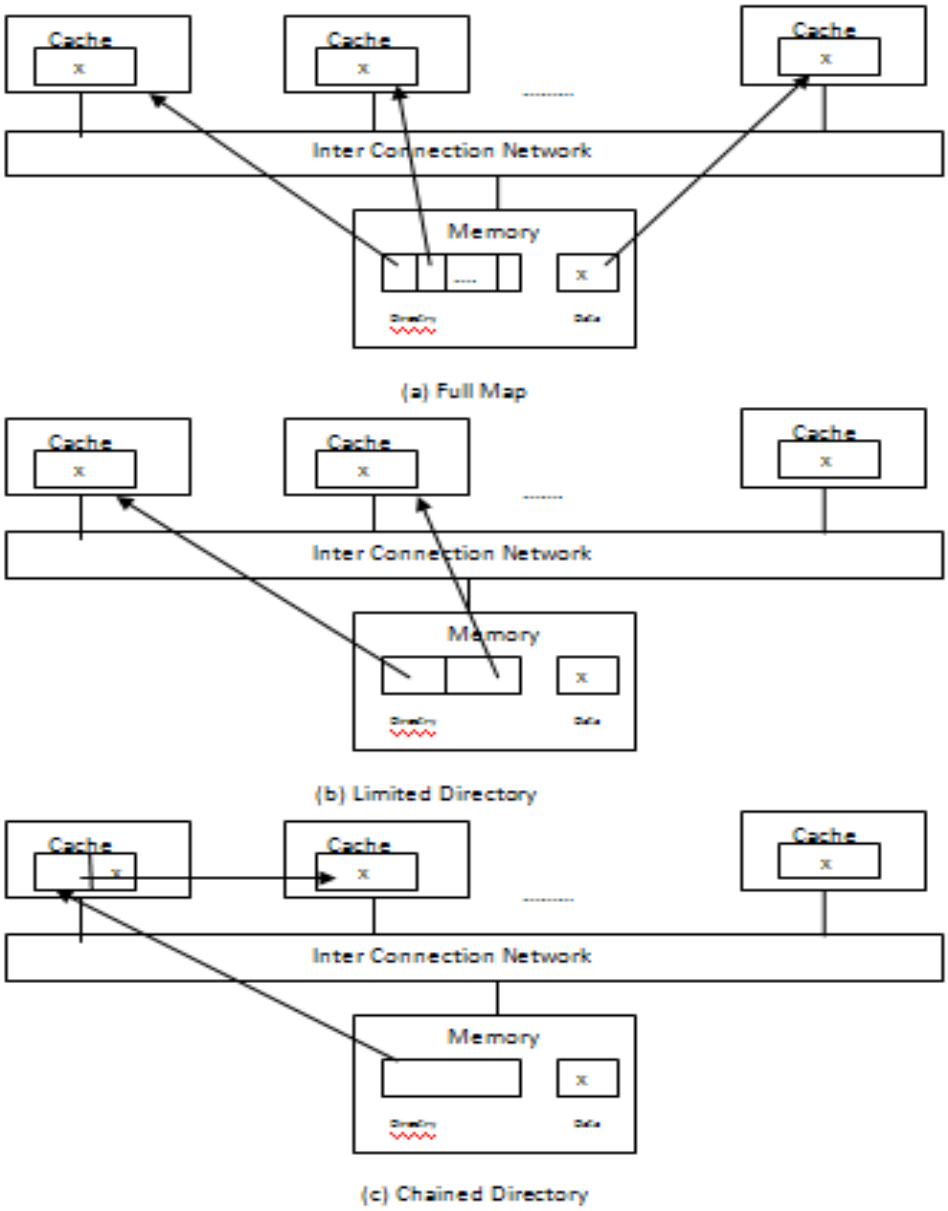



**Figure 3.6: Three types of Directory Organization**

### 3.2.2.1 Full mapped Directory Protocol

The main characteristic of these schemes is that the directory is stored in the main memory, and contains entries for each memory block. An entry points to exact locations of every cached copy of memory block, and keeps its status. Using this information, coherence of data in private

caches is maintained by directed messages to known locations, avoiding usually expensive broadcasts, burden for maintaining correct value of this bit. The main advantage of the full-map approach is that locating necessary cached copies is easy, and only caches with valid copies are involved in coherence actions for a particular block Because of that, they deliver best performance of all directory schemes. But there are some of disadvantages. Centralized controller is inflexible for system expansion by adding new processors. Also, these schemes are not scalable for several reasons. Since all requests are directed to the central directory, it can become a performance bottleneck. The most serious problem is significant memory overhead for full-map directory storage in multiprocessor systems with a large number of processors. One approach to alleviate this problem is to reduce directory size.

### 3.2.2.2. Limited Directory Protocol

To cope-up with the problem of memory overhead in full-map directory schemes led to centralized schemes with partial maps or limited directories. They replace the presence bit vector with a small number of identifier pointing to cached copies (Figure 4.2(b)). Size difference between a full mapped and a limited directory, for small i and large N, is significant (where i is the number of pointers and N is the number of processors). This concept is justified by findings of some studies that the number of simultaneously existing cached copies of the same block is usually small. Entries in limited directories contain a fixed number of pointers. Special actions have to be taken when the number of cached copies exceeds the number of pointers. The schemes with broadcast capability allow that situation, because they can invalidate all copies using a broadcast signal when necessary. If protocol disallows broadcasts, one copy has to be invalidated, to free the pointer for a new cached copy. These protocols put an upper limit on the number of simultaneously cached copies of a particular block. The scalability of limited

directory protocols, in terms of memory overhead for directory storage, their performance heavily depends on sharing characteristics of parallel applications.

### 3.2.2.3. Chained Directory Protocol

The other way to insure scalability of directory schemes, with respect to tag storage efficiency, is the of chained directory protocol. It is the most important that whatever approach we are using does not limit the number of cached copies of shared data block. Entries of such a directory are organized in the form of linked lists, where all caches sharing the same block are chained through pointers into one list (Figure 3.6(c)). Unlike the full mapped directory and limited directory approaches, chained directory approach is spread across the individual caches. Entry in main memory is used only to point to the head of the list and to keep the shared data block status. Requests for the block are issued to the memory, and subsequent command through the list, using the pointers. The chained directories can be organized in the form of either singly or doubly linked lists.

The main advantage of chained directory approach is their scalability; its performance is almost as good as in full-map directory schemes. Because of better handling of the replacement situation, doubly linked lists perform slightly better compared to singly linked lists, at the expense of being more complex and using twice as much storage for pointers

## 3.2.3. Hybrid Cache Coherence Protocol

As we know that different data blocks may exhibit different types of access behavior, a system which uses more than one cache coherence protocol has the potential to lead to an improvement in performance. Using the appropriate protocol can lead to a reduction in cache misses and

coherence traffic which can result in performance improvement. A hybrid cache coherence protocol can use any one of the basic protocols, such as invalidate or update for each cache block.

In addition to this the data access behavior for a particular cache block may change during the execution of an application. Hybrid protocols are also known as competitive-update protocols which are still suboptimal for migratory data. Migratory data is data that is read, modified, and written by many processors in turn. WI (Write invalidate) by itself is better for migratory data, as updates are wasted for migratory data. To handle migratory data in competitive-update protocols in better way the migratory data is dynamically detected in With WI (Write invalidate), a read request is sent to the home site of the data, and then an invalidate message is sent when the data is updated. Since the processor knows that it will both read and write migratory data, these two messages can be combined into a read-exclusive request.

The protocol which is being used at a certain time during the execution is determined by a decision function, which can be implemented in hardware or software. The ultimate goal of this function is to change the protocol for each cache block at an appropriate time to improve the performance of the system. The function can be based on some heuristic to reduce the amount of traffic generated or latency. If the decision function is not accurate and makes a wrong decision, then it can increase traffic. Various dynamic hybrid cache coherence protocols have been proposed and implemented. They differ mainly in the implementation of the decision function and the amount of hardware support provided for alternate protocols. The decision function can select the appropriate protocol prior to or during the execution of an application.

We can classify decision function as:

- Online Decision Function
- Offline Decision Function

To further increase the potential for performance improvement, the protocol for a block can be changed during the execution of an application. These protocols are known as dynamic or adaptive hybrid cache coherence protocols.

### Lock Based Protocol

The recent improvement on directory-based protocols is Lock based protocol. This protocol promises to be more scalable than directory-based coherence by implementing scope consistency. Scope consistency is a compromise between lazy release consistency and entry consistency. When processor P acquires lock L from processor Q, it does not pick up all modifications that have been visible to Q as in lazy release consistency. Rather, it picks up only those modifications written by Q in the critical section protected by L.

In this protocol there is no directory and all coherence actions are taken through reading and writing notices to and from the lock which protects the shared memory. The lock release sends all write notices to the home of the lock and all modified memory lines. On lock acquire, the processor knows from the lock's home which lines have been modified and can retrieve modifications. A processor is stalled until all previous acquire events have been performed. This method is more scalable because directory hardware is not needed, although lock release may be slow as a processor must wait for all writes to be transmitted and receive the acknowledgements.

## 3.3 Summary

In this chapter we have thoroughly surveyed all the existing software and hardware based cache coherence approaches. After their analysis we can conclude that though software based cache

coherence protocols are cost effective but if we compare their performance with hardware based protocols then it is very low. But again with hardware based cache coherence protocols cost factor is getting increased. Lots of work can be done over software based protocols.

# Chapter 4: Proposed Idea and Related Work

## 4.1 Introduction:

Another aspect of the MSI model susceptible to be improved is the following: a cache with a block in modified state does not distinguish between an exclusive copy that has been modified and an unmodified exclusive copy that is only held by that cache (since any other cache does not currently have a valid copy). This situation can lead to unnecessary data traffic, as the replacement of unmodified exclusive blocks cause the blocks to be written back to main memory. Besides this problem, another concern arises when the MSI model is used in a multiprocessor running a sequential application. In this case, when a processor reads in and modifies a memory block, the MSI model generates two consecutive cache misses (even though there are no sharers), since the first cache miss retrieves the block in shared state and the second it is necessary to convert S state to M state.

The two aforementioned situations are avoided by adding a state indicating that the block is the only (exclusive) copy but it is not modified. This new state, called *exclusive* (E), indicates an intermediate level of binding between shared and modified. It is exclusive, so unlike the shared state, the cache can perform a write (directly transitioning to the modified state). However, the exclusive state does not imply ownership (memory has a valid copy), so unlike the modified state, the cache does not need to reply when observing a request upon the block. Variants of this MESI protocol [95] are used in many microprocessors, including the Intel Pentium, PowerPC 601, and the MIPS R4400 used in the Silicon Graphics Challenge multiprocessors.

MESI protocol [11] also referred to as Illinois protocol due to its development at the University of Illinois at Urbana-Champaign is very renowned cache coherence protocol and it supports

write-back cache. In this protocol every cache line can have one of the following states (as shown in figure 4.1): Modified, Exclusive, Shared, Invalid. There explanation is given below:

Modified: In this the cache line is present only in the current cache, and hence is dirty; also it's been modified from the value available in main memory. So, the cache is required to write the data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state. The write-back will change the state of the line to the Exclusive state.

Exclusive: The cache line is present only in the current cache, but is clean; it matches main memory. It may be changed to the Shared state at any time, in response to a read request. Alternatively, it may be changed to the Modified state when performing write to it.

Shared: Indicates that this cache line may be stored in other caches of the machine as well and is "clean"; it matches the main memory. The line may be discarded (changed to the Invalid state) at any time.

Invalid: Indicates that the cache line is invalid. The detailed explanation is shown in figure 2.

It is better than MSI protocol [36] since for every write operation two transitions are performed, even when the data block taken into consideration is not shared. In the first transition it gets the memory block in the shared state and in second transition causes write, which also changes the state of that data block to shared state from modified state. It adds a new state to MSI protocol i.e. Exclusive state which reduces the traffic because of write operation performed for the shared data block [11].

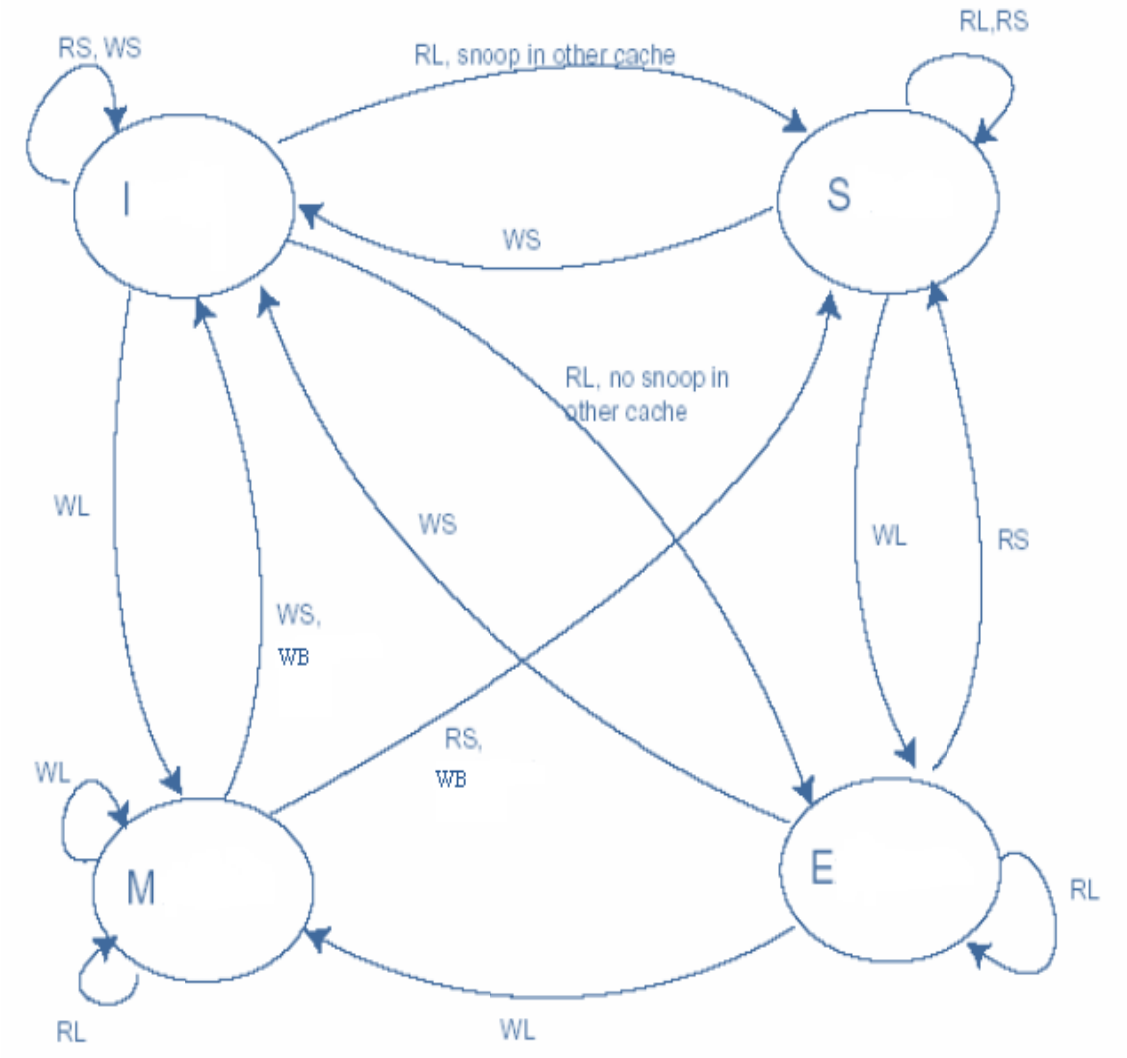



**Figure 4.1: Transition diagram of MESI protocol**

## 4.2 Proposed Idea:

After thoroughly studying the MESI (Modified, Exclusive, Shared and Invalid) protocol and the code of MARSSx86 (Micro Architectural and System Simulator) simulator, it was found that in the existing code of MESI (Modified, Exclusive, Shared and Invalid) protocol invalid to invalid transition still exists there. We thought if it can be modified or eliminated then it should increase the system performance by making this transition zero or reducing it at very high extent and also in terms of cycles per second and commits per second. We are actually trying to overcome this

deficiency of MESI (Modified, Exclusive, Shared and Invalid) protocol. Infact by doing this we have achieved success at great extent with dual cores (i.e. 2 cores), quad cores (i.e. 4 cores) and octet cores (i.e. 8 cores) on programs of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark viz. CANNEAL, FERRET and SWAPTIONS. We thought of actualizing this idea on both level 2 cache as Shared and level 2 cache as Private.

## 4.3 Related Work

The MARSSx86 (Micro Architectural and System Simulator) is a simulator that works only on 64-bit operating system and 64-bit processor. So first we have installed LINUX (UBUNTU 11.10) on the system. After successfully installing 64-bit operating system, we installed MARSSx86 simulator with the help of github. The www.github.com is a website used to share open source simulator code and via this we can easily update the code existing in the system if required. Though we confront lots of problems while compiling and running the MARSSx86 simulator but finally we conquered all the issues with lots of efforts. Then we initiated with the task of getting standard results i.e. stats file (as MARSS simulator output comes in term of stats file.), while taking these we kept level-2 cache as private once and at the second time we kept level 2 cache as shared.

We used PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark for getting all the standard and modified code results i.e. stats file. We keep on differing number of cores viz. 2 (dual cores), 4 (quad cores), and 8 (octet cores) for getting standard results and the results with modified code. After getting the standard results of our experiment on CANNEAL, FERRET and SWAPTIONS (programs of PARSEC benchmark) we proceeded with the actual task of modifying the code of MESI protocol. After extensive study of MARSSx86 simulator's code which is available without a hitch as open source, we have

transformed the code of file mesiLogic.cpp which contains the nitty-gritty of MESI protocol while executing with MARSSx86 simulator. In this file invalid to invalid transitions are present. We have modified the code precisely at this segment. Then again compiled and run this code with CANNEAL, FERRET and SWAPTIONS (programs of PARSEC benchmark) with the variation of number of cores i.e. (dual cores), 4 (quad cores), and 8 (octet cores). After getting experiment results in the form of stats files (results of experiments comes in the form of stats file) we have thoroughly compared them with the standard result's stats files. We have noticed that by keeping number of cores 2 i.e. dual core we have made invalid to invalid transition as zero successfully for all 3 programs of PARSEC benchmark viz. CANNEAL, FERRET and SWAPTIONS.

## 4.4 Summary:

In this chapter we have proposed an idea to overcome the existing deficiency in the code of MESI protocol of MARSS (Micro Architectural and System Simulator) simulator. In fact in further chapters we will notice that this proposed idea worked successfully by 99% for dual cores, quad cores and octet cores. Then we have mentioned the work done for getting results via MARSS simulator.

# Chapter 5: Evaluation Methodology

This chapter presents the simulation tools used for evaluating the performance and the relative behavior of our proposals. Since it is not to evaluate our proposals on all possible system configurations, we have performed a relative accurate comparison between the different approaches by using a simulation model of current multiprocessors.

MARSS provides a unique x86 a complete system simulation framework to simulate or emulate multiple cores systems running unmodified operating system, their libraries and their applications. The capability of MARSS to simulate various IO devices with a cycle-accurate processor model and collect region specific performance statistics makes MARSS crucial tool in complete system analysis. These features and ability to model heterogeneous core designs, presents MARSS as an attractive framework for evaluating design alternatives of future systems as well as different processor microarchitectures.

## 5.1 Introduction

Single and multiple cores processors implementing the x86 instruction set architecture (ISA) are deployed at many computing platforms today, beginning from high-end servers to desktops and ultimately down to mobile devices, including smart phone market segment and beyond. The one principle advantage of using the x86 processors in the complete range of the product spectrum is to facilitate the swift deployment of the wide variety of x86 application binaries. It is thus important to have a complete system simulation tool that comprises realistic simulation models for other systems level components such as the chipset, DRAM, network interface cards and peripheral devices in addition to accurate simulation models for single and multiple cores

processors implementing the x86 ISA. Such a tool is useful for evaluating and developing products that will use current and emerging single and multiple cores x86 chips. This diagram presents an open source full system simulation tool, called MARSS – **M**icro **Ar**chitectural and **S**ystem **S**imulator (shown in figure 5.1), which meets this crucial need. The x86 CPU simulation components of MARSS is based on an extended and modified version of PTLsim [36]. The specific features of MARSS are:

- MARSS uses a cycle-accurate simulation models for out-of-order and in-order single core and multiple cores CPUs implementing the x86 ISA. These are integrated into the QEMU [37] complete system emulation environment.
- MARSS supports seamless switching between the cycle-accurate simulation mode and the native x86 emulation mode of QEMU, allowing the fast-forwarding of simulation in the emulation mode to a region of interest where cycle-accurate simulation is needed.
- Unmodified operating systems can be booted on MARSS and the execution of unmodified x86 binaries of applications and existing libraries can be simulated on MARSS.
- MARSS includes cycle-accurate models of a contemporary memory hierarchy for single-core and multiple cores processor chips, including coherent caches and a DRAM memory system. Simulation speeds of 200 to 400 kilo instructions per second are realized in the cycle-accurate simulation mode of multiple cores processor chips[38].
- MARSS allows system-level data to be imported into the simulator from the underlying emulated system. This not only permits the use of realistic data but also enables users to judge the effect of core designs on the rest of the system and vice-versa.

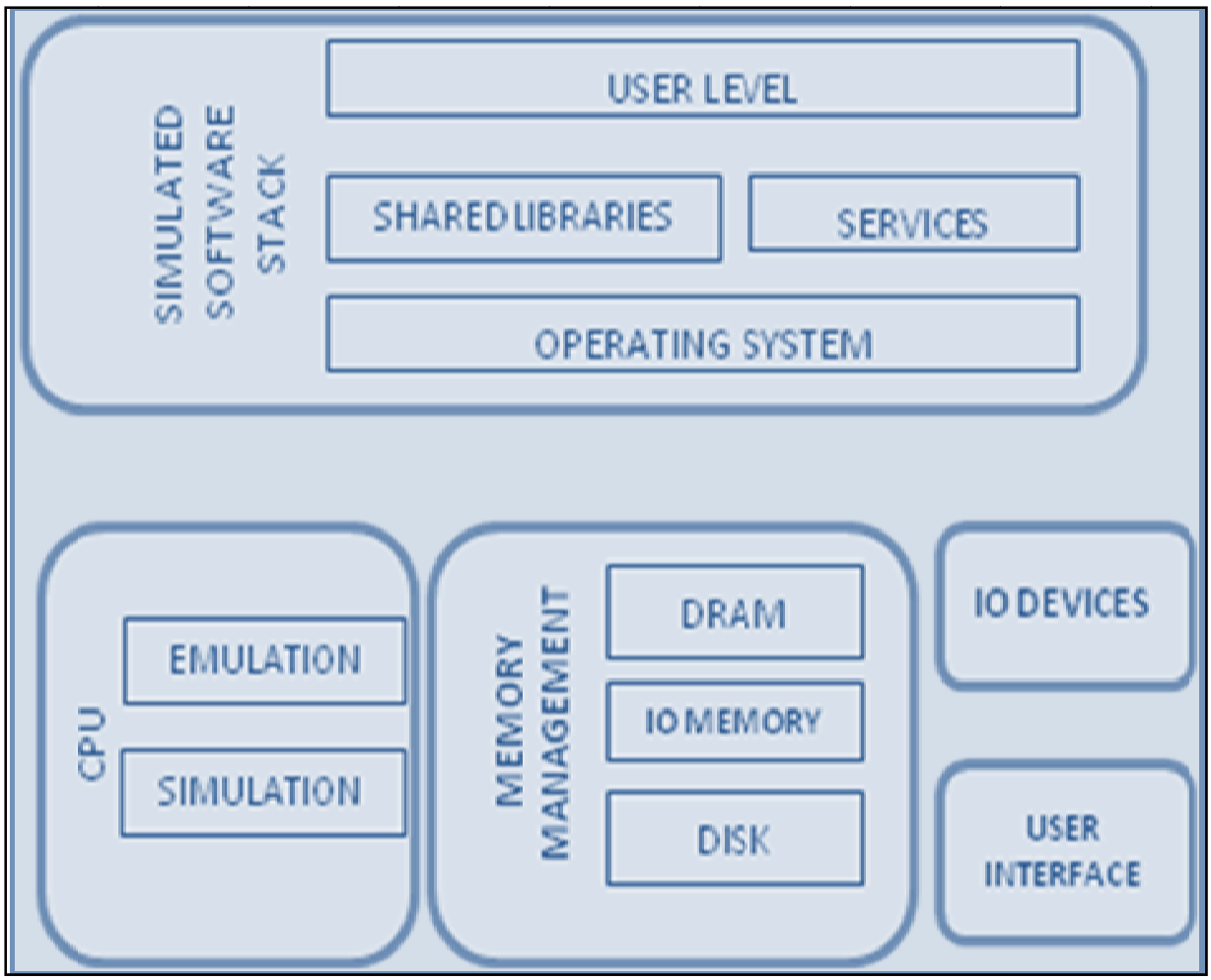



**Figure 5.1: The MARSS simulator framework**

The MPTLsim simulator [39] comes closest to MARSS in terms of functionality as a multiple cores simulator with coherent caches, but MPTLsim has many of the attributes of PTLsim, which requires modified hypervisors and root level privileges for running. MPTLsim, unlike MARSS, does not implement the other system components that QEMU provides, as MPTLsim uses the Xen virtualization facilities. There are many x86 ISA-based multiple cores simulators that are available in the public domain. These include FeS2 [40], a variant of M5 (announced but not released yet), Zesto [41], Bochs [42] and others. FeS2 uses an event-driven simulation framework, using the Ruby memory model of GEMS [43] and uses the x86 ISA-to-µop decoding

logic of PTLsim while requiring use of the Simics simulation framework. Zesto is implemented on top of the well-used Simplescalar simulator and primarily focuses on microarchitectural details of x86 implementations, supporting additional features that are not implemented in PTLsim or in its variants. Bochs [42] is a system-level emulator, capable of "booting" several operating systems and emulates only single core CPUs implementing the 32-bit x86 ISA.

## 5.2 Overview of MARSS

MARSS is a very unique cycle-accurate simulation framework built on top of QEMU's [37] solid and versatile emulation framework. QEMU's emulation framework consists of various components viz. a CPU emulator, memory management unit, IO devices, chipsets etc. Figure 1 shows a high-level view of various components of QEMU along with the added CPU simulation framework of MARSS.

## 5.3 Simulation Framework

MARSS uses PTLsim [36] a cycle-accurate simulator, as the basis of its CPU simulation environment on the top of QEMU. PTLsim provides a cycle-accurate simulation model for out-of-order (OOO) x86 CPUs, modeling the decomposition of x86 instructions into RISC-like μops and using basic block buffers to form traces of x86 μops, as in many real x86 implementations. PTLsim provides full system simulation capabilities by using the Xen Virtual Machine Framework and a modified Xen hypervisor. PTLsim was extensively modified to realize some key features of the MARSS simulation or emulation framework and extensive changes were done to port it to QEMU. The PTLsim substrate was also augmented to model multiple cores microprocessors with coherent caches, a DRAM memory system and interconnection, on-chip interconnections and to support MMX instructions. The original PTLsim OOO-core supports

SMT models where many core resources are shared among different threads and on-chip caches are shared among these threads. MARSS extends this design to model asymmetric heterogeneous cores and coherent caches.

## 5.4 Benchmark

The Princeton Application Repository for Shared-Memory Computers (PARSEC), a benchmark suite for studies of Multiprocessors and multiple cores. Previously available benchmarks for multiprocessors have focused on high-performance computing applications and used a limited number of synchronization methods. PARSEC includes emerging applications in recognition, mining and synthesis (RMS) as well as systems applications which behave as large-scale multi-threaded commercial programs. There characterization shows that this benchmark suite is diverse in working set, locality, data sharing, synchronization, and off-chip traffic. The benchmark suite is available to the public.

The applications are divided into three phases: an initial serial phase, a parallel phase, and a final serial phase. The parallel phase is called the region of interest (ROI) and is marked in the application source code by calls to the PARSEC hooks library. The hooks library can be used to perform certain actions upon entering and leaving the ROI.

There are certain programs in PARSEC benchmark viz. CANNEAL, FERRET, SWAPTIONS, RAYTRACE, BODYTRACK, BLACKSHOLES, DEDUP, FACESIM, FLUIDANIMATE, FRQMINE, STREAMCLUSTER and VIPS. We have done experiments on 3 programs of PARSEC benchmark viz. CANNEAL, FERRET and SWAPTIONS whose description is mentioned in this chapter.

### 5.4.1 CANNEAL Overview

- Minimizes the routing cost of a chip design with cache-aware simulated annealing
- Electronic Design Automation (EDA) kernel (Princeton)
- Input is a synthetic netlist
- Fine-granular parallelism, no problem decomposition
- Uses atomic instructions to synchronize
- Synchronization strategy based on data race recovery rather than avoidance
- Huge working sets, communication intensity only constrained by cache capacity.

### 5.4.2 SWAPTIONS Overview

- Prices a portfolio of SWAPTIONS with the Heath-Jarrow-Morton framework
- Computational finance application (Intel)
- Input is a portfolio of derivatives
- Coarse-granular parallelism, static load-balancing
- Medium-sized working sets, little communication

### 5.4.3 FERRET overview:

- Search engine which finds a set of images similar to a query image by analyzing their contents
- Server application for content-based similarity search of feature-rich data (Princeton)
- Input is an image database and a series of query images
- Pipeline parallelism with multiple thread pools
- Huge working sets, very communication intensive

## 5.5 Summary

In this chapter we have thoroughly studied about the MARSS (Micro Architectural and System Simulator) simulator which is an open source and we have studied about PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark whose three programs viz. CANNEAL, FERRET and SWAPTIONS are used for getting experimental results.

# Chapter 6: Experimental Results

For taking results on MARSSx86 (Micro Architectural and System Simulator) simulator I have taken PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark. We have varied the configuration with number of cores and with the status of LEVEL 2 cache as PRIVATE and SHARED. For achieving these experimental results I have taken number of cores as 2 (Dual Cores), 4 (Quad Cores), and 8 (Octet Cores). I have taken results on CANNEAL, FERRET and SWAPTION program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark. The results come in the form of stats file of (Micro Architectural and System Simulator) simulator. The standard results are compared with the results of modified code and its comparison is shown in sections given below.

## 6.1. Results while keeping LEVEL 2 Cache as private:

### 6.1.1. Results with number of cores as 2:

While keeping number of cores as two we noticed that we have completely eliminated invalid to invalid transitions with CANNEAL, FERRET and SWAPTIONS programs of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark. We have kept level 2 cache as private in this section.

#### 6.1.1.1. Results with CANNEAL:

The simulation result of CANNEAL program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark has shown that we have not only made invalid to invalid transitions zero (as shown in table 6.1), but also enhanced performance in other concerns

too. The cycles per second is increased at kernel and user level. Infact the commits per second are also increased.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| CANNEAL | 2349 | 0 |
| | 12705 | 0 |
| | 5376 | 0 |
| | 29079 | 0 |

**Table 6.1: Invalid to Invalid transitions with CANNEAL taking 2 cores with L2 private**

The graph of figure 6.1 is also showing the variation of standard results and results with modified codes. There is a huge height difference between result of the experiment of standard code and result of the experiment of modified code of MESI protocol.

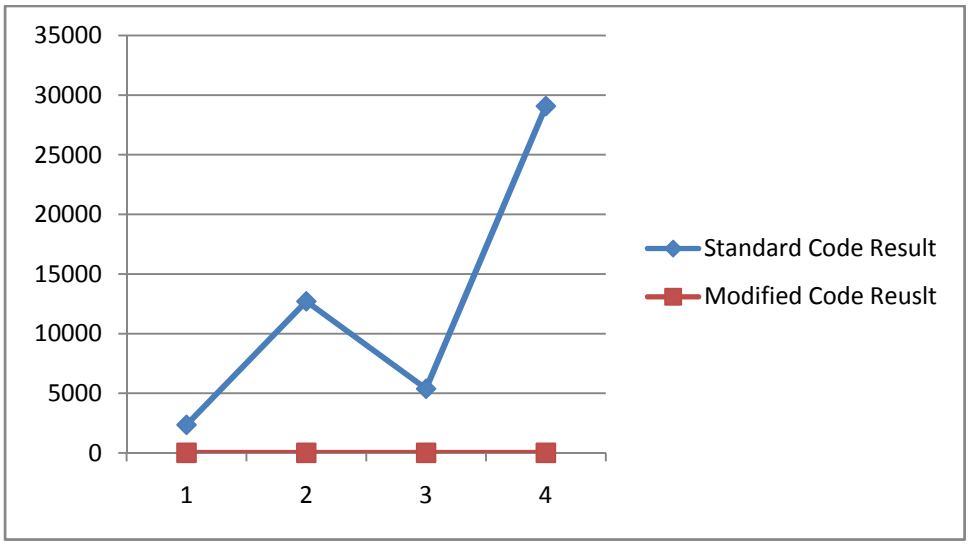



**Figure 6.1: Invalid to Invalid transitions with CANNEAL taking 2 cores with L2 private**

### 6.1.1.2. Results with FERRET:

The simulation results of FERRET program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark are presenting that I have not just reduced invalid to invalid transitions to zero (as shown in table 6.2), but also increased the system performance in other concerns too. The cycles per second are increased at kernel and user level.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| FERRET | 1288 | 0 |
| | 7550 | 0 |
| | 90 | 0 |
| | 21096 | 0 |

**Table 6.2: Invalid to Invalid transitions with FERRET taking 2 cores with L2 private**

The graph of figure 6.2 is also showing the variation of standard results and the results with modified codes for FERRET program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark. There is zero in place of modified code invalid to invalid transition.

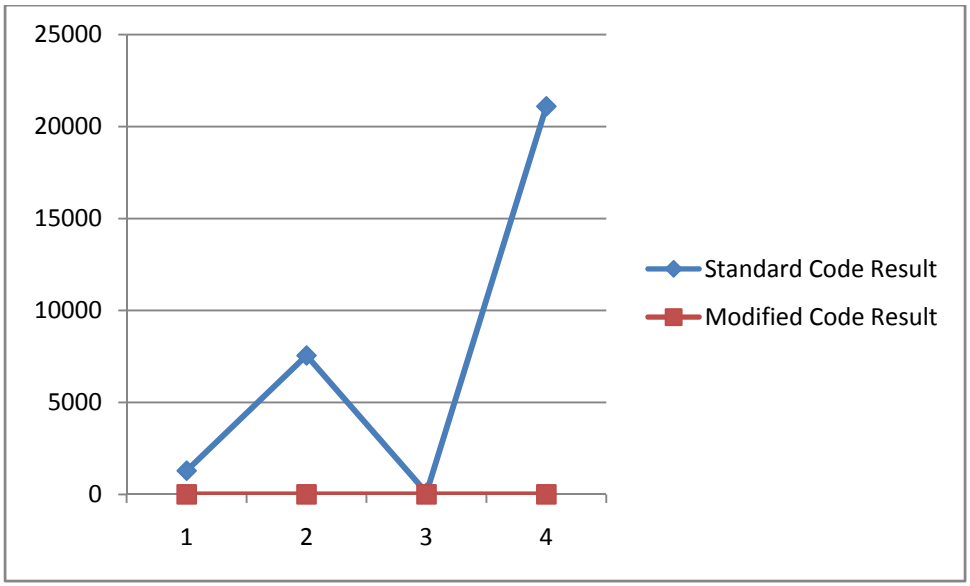



**Figure 6.2: Invalid to Invalid transitions with FERRET taking 2 cores with L2 private**

### 6.1.1.3. Results with SWAPTIONS:

The simulation results of SWAPTIONS program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark are showing that not only invalid to invalid transitions are reduced to zero (as shown in table 6.3), but also cycles per second are increased and its also helping in enhancing the system performance.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| SWAPTIONS | 2485 | 0 |
| | 3736 | 0 |
| | 403 | 0 |
| | 613 | 0 |

**Table 6.3: Invalid to Invalid transitions with SWAPTIONS taking 2 cores with L2 private**

The graph of figure 6.3 is also showing the variation of standard results and the results with modified codes for SWAPTIONS program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark. There is zero in place of modified code invalid to invalid transition.

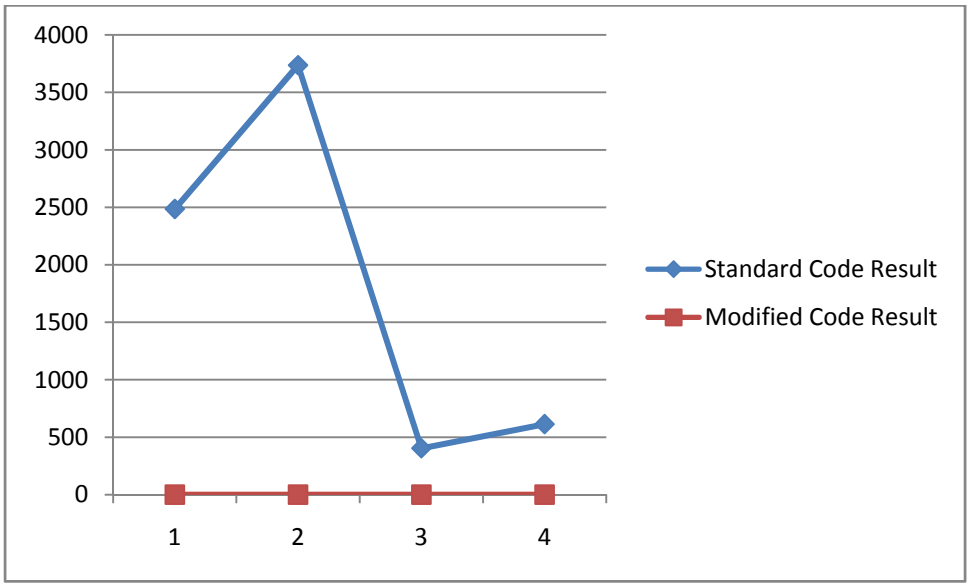



**Figure 6.3: Invalid to Invalid transitions with SWAPTIONS taking 2 cores with L2 private**

## 6.1.2. Results with number of cores as 4:

Now I kept number of cores as 4 i.e. quad cores are during simulation on CANNEAL, FERRET and SWAPTIONS programs of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark. Now again after analyzing the results of actual code of MESI protocol and modified code on MESI protocol we concluded that success in reducing invalid to invalid transitions is achieved at very high extent as mentioned in further sections.

### 6.1.2.1. Results with CANNEAL:

The simulation result of CANNEAL program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark has shown that I have achieved success till 99% in reducing invalid to invalid transitions at the great extent, as shown in table 6.4. Infact cycles per second and commits per second are increased at kernel and user level.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| CANNEAL | 84305 | 1748 |
| | 48865 | 4031 |
| | 74570 | 8549 |
| | 56536 | 1507 |

**Table 6.4: Invalid to Invalid transitions with CANNEAL taking 4 cores with L2 private**

The graph of figure 6.4 is also showing the variation of the standard results and the results with modified codes. There is a huge difference between result of the experiment of standard code and result of the experiment of modified code of MESI protocol.

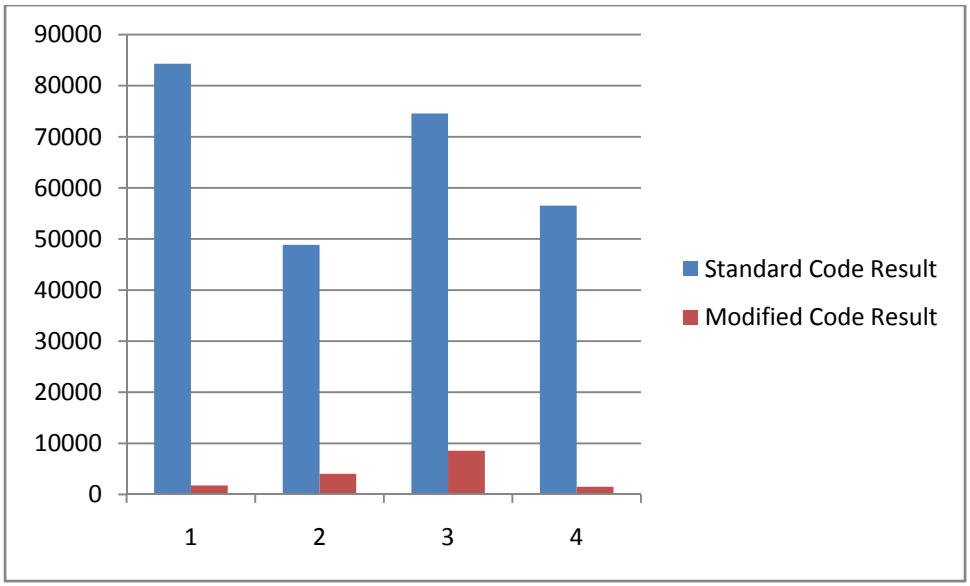



**Figure 6.4: Invalid to Invalid transitions with CANNEAL taking 4 cores with L2 private**

### 6.1.2.2. Results with FERRET:

The simulation results of FERRET program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark are representing that invalid to invalid transitions are depreciated around 99% as shown in table 6.5. I have not just reduced invalid to invalid transitions to zero (as shown in table 6.5), but also increased the system as cycles per second and commits per seconds are increased at kernel and user level.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| FERRET | 79300 | 2658 |
| | 15894 | 2826 |
| | 6680 | 1215 |

| | | |
|---|---|---|
| | 79802 | 24664 |

**Table 6.5: Invalid to Invalid transitions with FERRET taking 4 cores with L2 private**

The graph of figure 6.5 is also showing the variation of standard results and the results with modified codes for FERRET program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark.

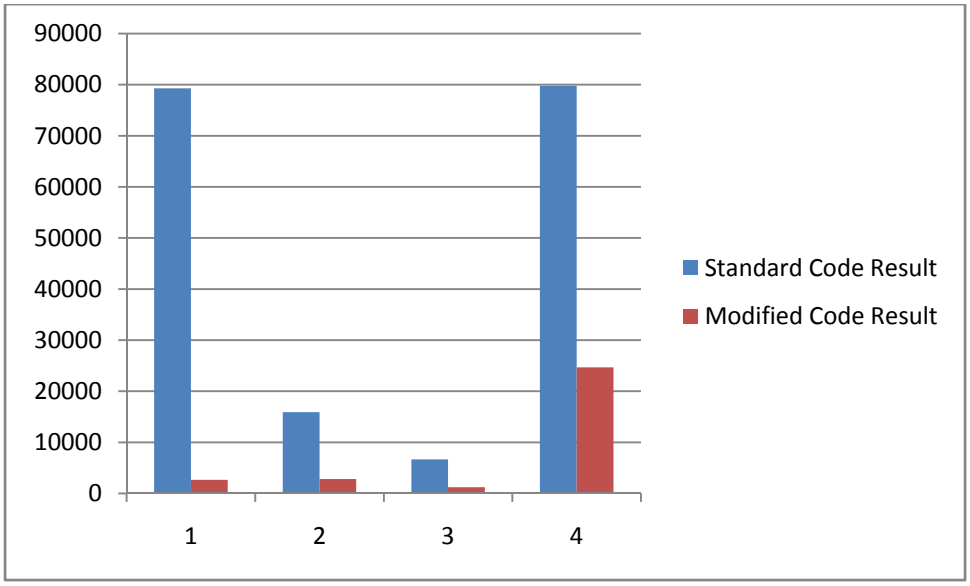



**Figure 6.5: Invalid to Invalid transitions with FERRET taking 4 cores with L2 private**

### 6.1.2.3. Results with SWAPTIONS:

The simulation results of SWAPTIONS program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark are showing that not only invalid to invalid

transitions are reduced by 70% (as shown in table 6.6), but also cycles per second are increased which is enhancing the system performance.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| SWAPTIONS | 6758 | 2070 |
| | 69371 | 24058 |
| | 72187 | 25127 |
| | 80460 | 30244 |

**Table 6.6: Invalid to Invalid transitions with SWAPTIONS taking 4 cores with L2 private**

The graph of figure 6.6 is also showing the variation of standard results and the results with modified codes for SWAPTIONS program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark.

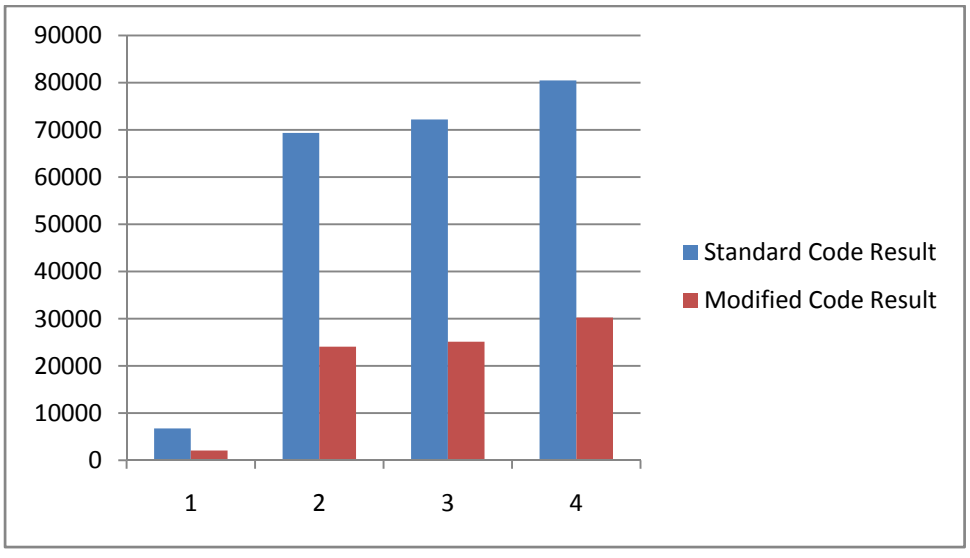



**Figure 6.6: Invalid to Invalid transitions with SWAPTIONS taking 4 cores with L2 private**

### 6.1.3. Results with number of cores as 8:

Now I kept number of cores as 8 i.e. octet cores are during simulation on CANNEAL, FERRET and SWAPTIONS programs of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark. Now again after analyzing the results of actual code of MESI protocol and modified code on MESI protocol we concluded that success in reducing invalid to invalid transitions is achieved at very high extent as mentioned in further sections.

#### 6.1.3.1. Results with CANNEAL:

The simulation result of CANNEAL program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark has shown that I have achieved success till 99% in reducing invalid to invalid transitions at the great extent, as shown in table 6.7. Infact cycles per second and commits per second are increased at kernel and user level.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| CANNEAL | 233495 | 2134 |
| | 217475 | 2027 |
| | 343174 | 42124 |
| | 324403 | 44442 |

**Table 6.7: Invalid to Invalid transitions with CANNEAL taking 8 cores with L2 private**

The graph of figure 6.7 is also showing the variation of the standard results and the results with modified codes. There is a huge difference between result of the experiment of standard code and result of the experiment of modified code of MESI protocol.

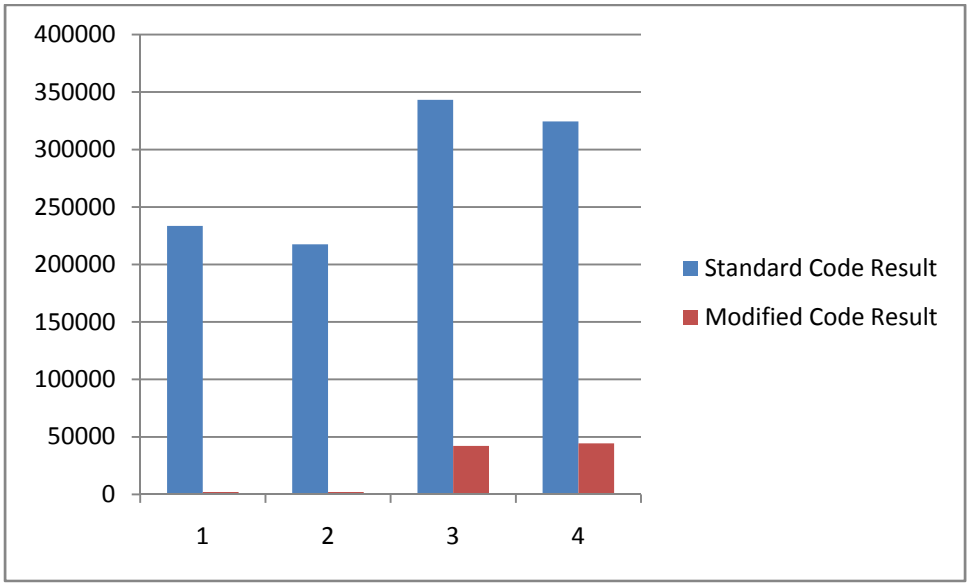



**Figure 6.7: Invalid to Invalid transitions with CANNEAL taking 8 cores with L2 private**

### 6.1.3.2. Results with FERRET:

The simulation results of FERRET program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark are representing that invalid to invalid transitions are depreciated around 99% as shown in table 6.8. I have not just reduced invalid to invalid transitions to zero (as shown in table 6.8), but also increased the system as cycles per second are increased at kernel and user level.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| FERRET | 16900 | 1844 |
| | 11162 | 1705 |
| | 15806 | 1920 |
| | 19059 | 2669 |

**Table 6.8: Invalid to Invalid transitions with FERRET taking 8 cores with L2 private**

The graph of figure 6.8 is also showing the variation of standard results and the results with modified codes for FERRET program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark.

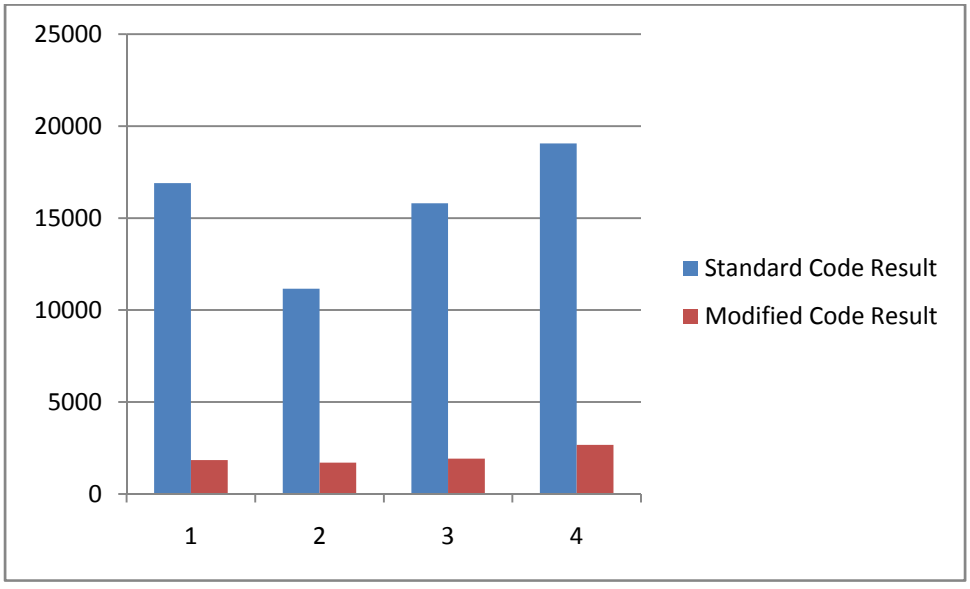



**Figure 6.8: Invalid to Invalid transitions with FERRET taking 8 cores with L2 private**

### 6.1.2.3. Results with SWAPTIONS:

The simulation results of SWAPTIONS program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark are showing that not only invalid to invalid transitions are reduced by 70% (as shown in table 6.9), but also cycles per second are increased which is enhancing the system performance.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| SWAPTIONS | 85713 | 28196 |
| | 83509 | 33813 |
| | 105853 | 38416 |
| | 191566 | 66612 |

**Table 6.9: Invalid to Invalid transitions with SWAPTIONS taking 8 cores with L2 private**

The graph of figure 7.9 is also showing the variation of standard results and the results with modified codes for SWAPTIONS program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark.

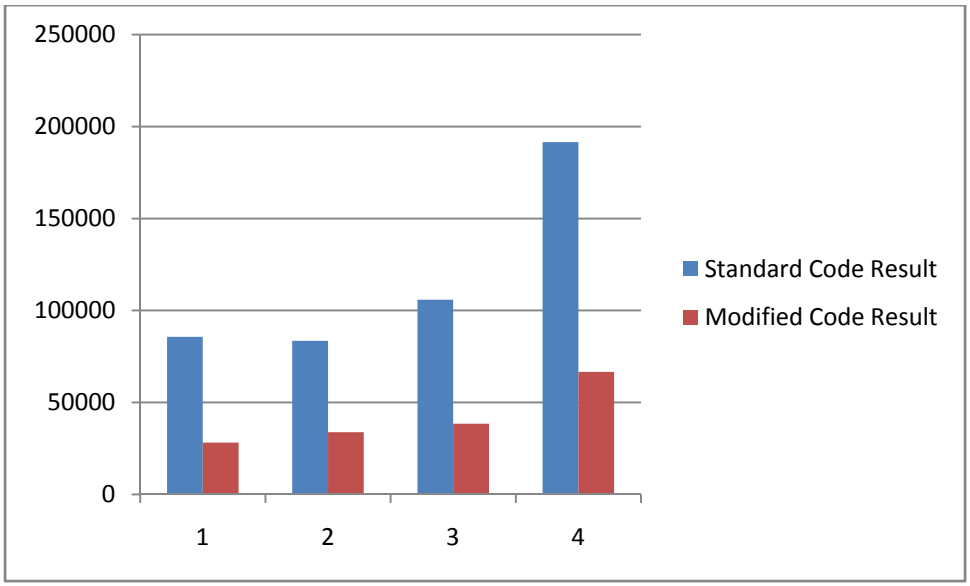



**Figure 6.9: Invalid to Invalid transitions with SWAPTIONS taking 8 cores with L2 private**

## 6.2. Results while keeping LEVEL 2 Cache as Shared:

### 6.2.1. Results with number of cores as 2:

While keeping number of cores as two we noticed that I am successful in completely eliminating the invalid to invalid transitions with CANNEAL, FERRET and SWAPTIONS programs of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark at level 1 data cache. I have taken level 2 cache as shared in this section.

#### 6.2.1.1. Results with CANNEAL:

The simulation result of CANNEAL program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark has shown that we have not only made invalid to invalid transitions zero (as shown in table 6.10) at level 1 data cache, but also enhanced performance in other concerns too. The cycles per second is increased at both kernel and user level.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| CANNEAL | 3179 | 0 |
| | 3932 | 0 |
| | 114 | 0 |
| | 171 | 0 |

**Table 6.10: Invalid to Invalid transitions with CANNEAL taking 2 cores with L2 shared**

The graph of figure 6.10 is also showing the variation of standard results and results with modified codes. There is a huge difference between result of the experiment of standard code and result of the experiment of modified code of MESI protocol.

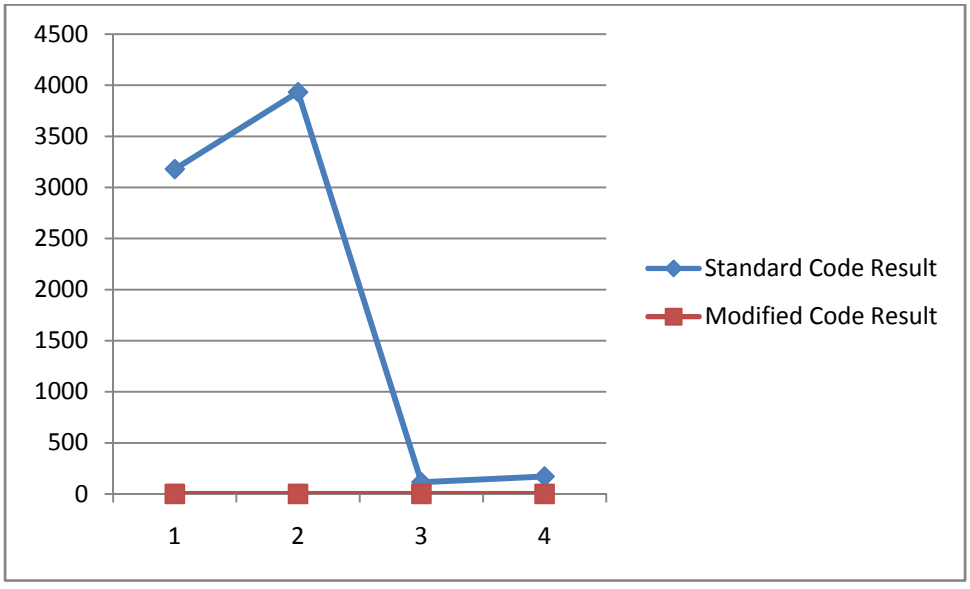



**Figure 6.10: Invalid to Invalid transitions with CANNEAL taking 2 cores with L2 shared**

### 6.2.1.2. Results with FERRET:

The simulation results of FERRET program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark are presenting that I have not just reduced invalid to invalid transitions to zero (as shown in table 6.11), but also increased the system performance in other concerns too. The cycles per second are increased at kernel and user level.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| FERRET | 118 | 0 |
| | 242 | 0 |
| | 969 | 0 |
| | 3279 | 0 |

**Table 6.11: Invalid to Invalid transitions with FERRET taking 2 cores with L2 shared**

The graph of figure 2 is also showing the variation of standard results and the results with modified codes for FERRET program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark. There is zero in place of modified code invalid to invalid transition.

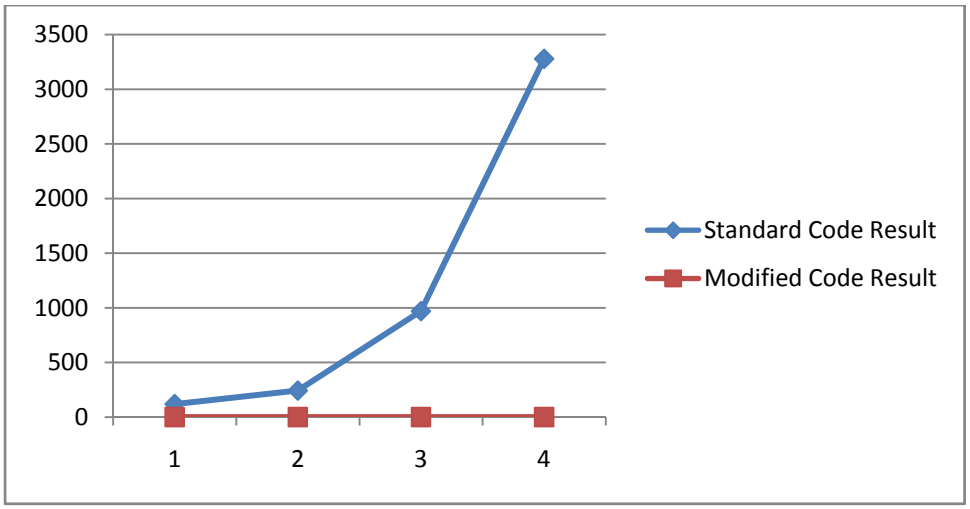



**Figure 6.11: Invalid to Invalid transitions with FERRET taking 2 cores with L2 shared**

### 6.2.1.3. Results with SWAPTIONS:

The simulation results of SWAPTIONS program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark are showing that not only invalid to invalid transitions are reduced to zero (as shown in table 6.12), but also cycles per second are increased and its also helping in enhancing the system performance.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| SWAPTIONS | 132 | 0 |
| | 83 | 0 |
| | 2113 | 0 |
| | 1962 | 0 |

**Table 6.12: Invalid to Invalid transitions with SWAPTIONS taking 2 cores with L2 shared**

The graph of figure 6.12 is also showing the variation of standard results and the results with modified codes for SWAPTIONS program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark. There is zero in place of modified code invalid to invalid transition.

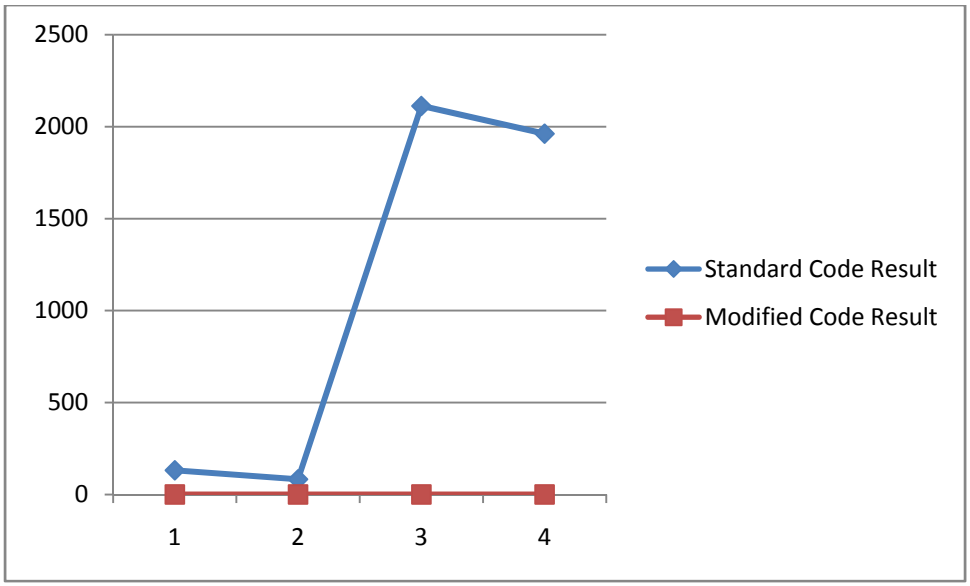



**Figure 6.12: Invalid to Invalid transitions with SWAPTIONS taking 2 cores with L2 shared**

## 6.2.2. Results with number of cores as 4

### 6.2.2.1. Results with CANNEAL:

The simulation result of CANNEAL program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark has shown that I have achieved success till 99% in reducing invalid to invalid transitions at the great extent, as shown in table 6.13. Infact commits per second are increased at kernel and user level.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| CANNEAL | 4994 | 340 |
| | 3537 | 298 |
| | 13500 | 307 |
| | 2068 | 295 |

**Table 6.13: Invalid to Invalid transitions with CANNEAL taking 4 cores with L2 shared**

The graph of figure 6.13 is also showing the variation of the standard results and the results with modified codes. There is a huge difference between result of the experiment of standard code and result of the experiment of modified code of MESI protocol.

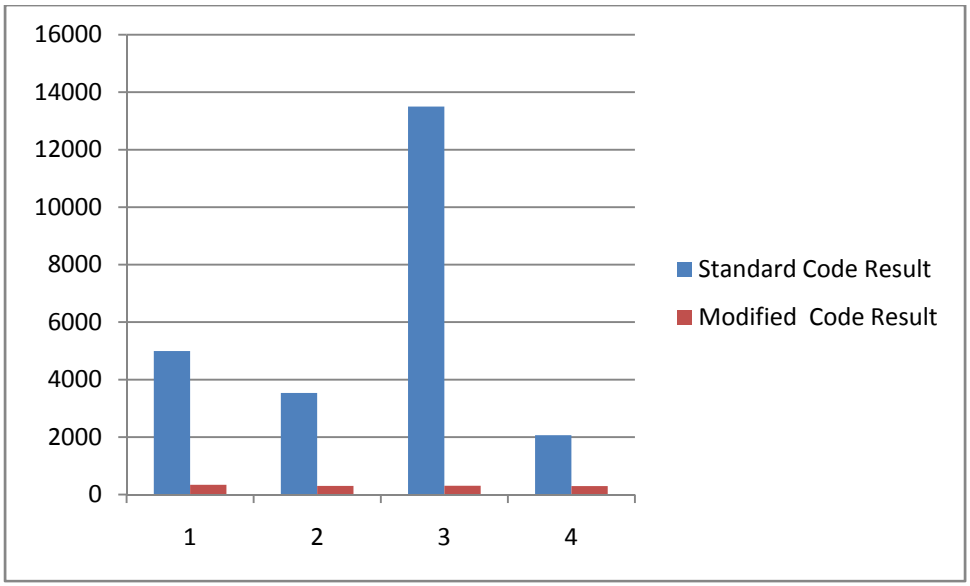



**Figure 6.13: Invalid to Invalid transitions with CANNEAL taking 4 cores with L2 shared**

### 6.2.2.2. Results with FERRET:

The simulation results of FERRET program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark are representing that invalid to invalid transitions are depreciated around 99% as shown in table 6.14. I have not just reduced invalid to invalid transitions to zero (as shown in table 6.14), but also increased the commits per seconds are increased at kernel and user level.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| FERRET | 8489 | 397 |
| | 15320 | 167 |
| | 6898 | 261 |
| | 1262 | 261 |

**Table 6.14: Invalid to Invalid transitions with FERRET taking 4 cores with L2 shared**

The graph of figure 6.14 is also showing the variation of standard results and the results with modified codes for FERRET program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark.

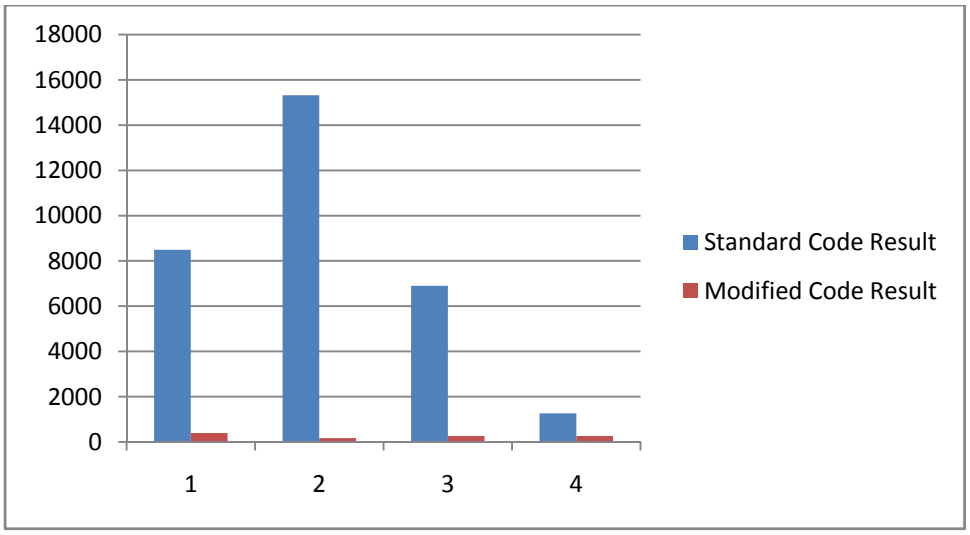



**Figure 6.14: Invalid to Invalid transitions with FERRET taking 4 cores with L2 shared**

### 6.2.2.3. Results with SWAPTIONS:

The simulation results of SWAPTIONS program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark are showing that not only invalid to invalid transitions are reduced by 70% (as shown in table 6.15), but also commits per second are increased which is enhancing the system performance.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| SWAPTIONS | 3303 | 232 |
| | 2183 | 268 |
| | 2976 | 164 |
| | 2332 | 170 |

**Table 6.15: Invalid to Invalid transitions with SWAPTIONS taking 4 cores with L2 shared**

The graph of figure 6.15 is also showing the variation of standard results and the results with modified codes for SWAPTIONS program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark.

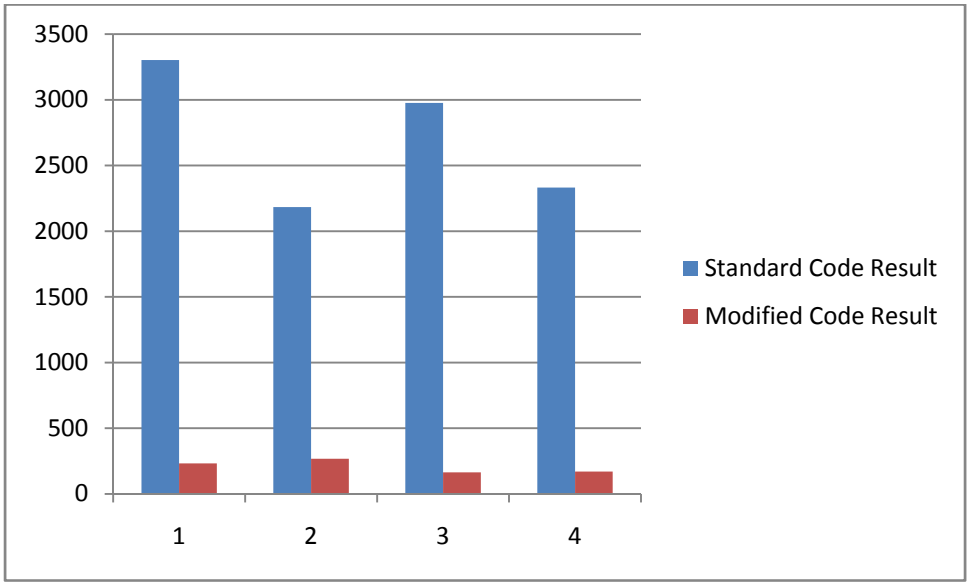



**Figure 6.15: Invalid to Invalid transitions with SWAPTIONS taking 4 cores with L2 shared**

### 6.2.3. Results with number of cores as 8:

Now I kept number of cores as 8 i.e. octet cores are during simulation on CANNEAL, FERRET and SWAPTIONS programs of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark. Now again after analyzing the results of actual code of MESI protocol and modified code on MESI protocol we concluded that success in reducing invalid to invalid transitions is achieved at very high extent as mentioned in further sections.

#### 6.2.3.1. Results with CANNEAL:

The simulation result of CANNEAL program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark has shown that I have achieved success till 99% in

reducing invalid to invalid transitions at the great extent, as shown in table 6.16. Infact cycles per second and commits per second are increased at kernel and user level.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| CANNEAL | 13346 | 841 |
| | 4468 | 834 |
| | 3317 | 448 |
| | 9845 | 57 |

**Table 6.16: Invalid to Invalid transitions with CANNEAL taking 8 cores with L2 shared**

The graph of figure 6.16 is also showing the variation of the standard results and the results with modified codes. There is a huge difference between result of the experiment of standard code and result of the experiment of modified code of MESI protocol.

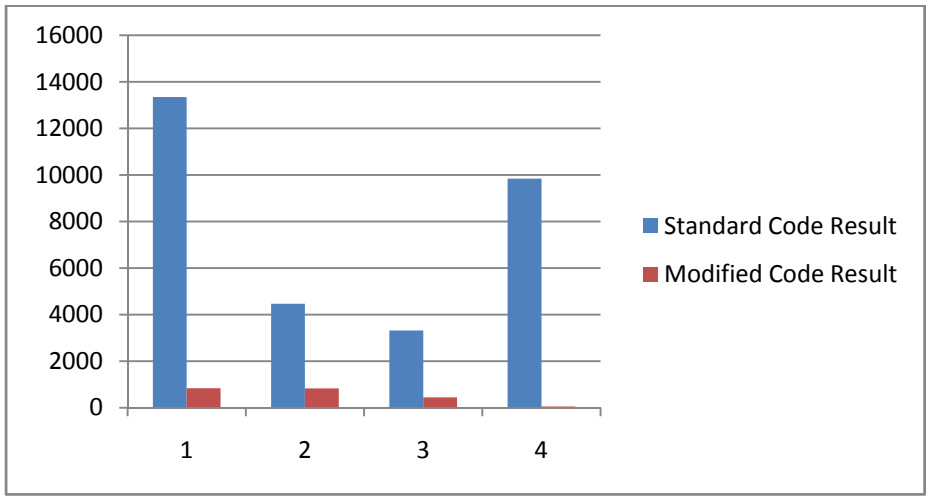



**Figure 6.16: Invalid to Invalid transitions with CANNEAL taking 8 cores with L2 shared**

### 6.2.3.2. Results with FERRET:

The simulation results of FERRET program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark are representing that invalid to invalid transitions are depreciated around 99% as shown in table 6.17. I have not just reduced invalid to invalid transitions to zero (as shown in table 6.17), but also increased the system as commits per second are increased at kernel and user level.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| FERRET | 9151 | 466 |
| | 3447 | 328 |
| | 10893 | 285 |
| | 32800 | 327 |

**Table 6.17: Invalid to Invalid transitions with FERRET taking 8 cores with L2 shared**

The graph of figure 6.17 is also showing the variation of standard results and the results with modified codes for FERRET program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark.

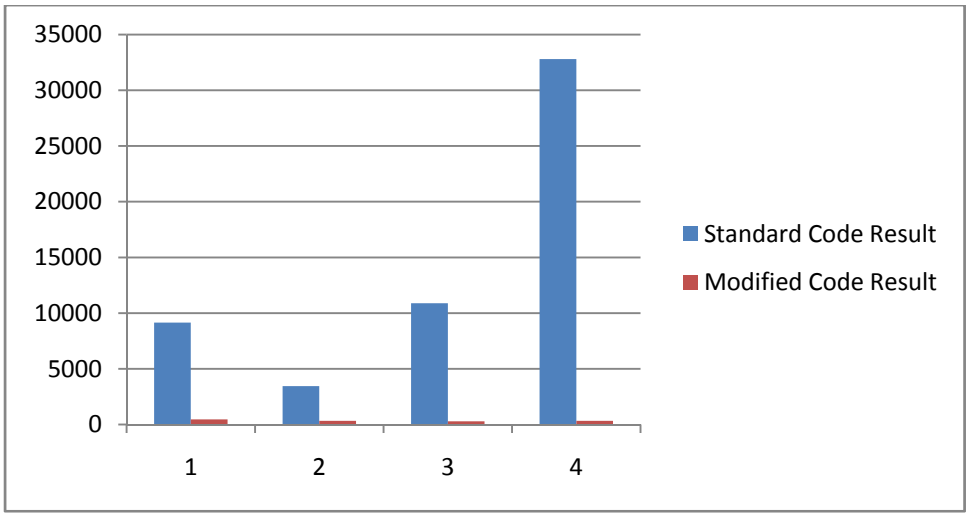



**Figure 6.17: Invalid to Invalid transitions with FERRET taking 8 cores with L2 shared**

### 6.2.3.3. Results with SWAPTIONS:

The simulation results of SWAPTIONS program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark are showing that not only invalid to invalid transitions are reduced by 70% (as shown in table 6.18), but also cycles per second are increased which is enhancing the system performance.

| Benchmark | Invalid to Invalid transition in Actual Code | Invalid to Invalid transition in Modified Code |
|---|---|---|
| SWAPTIONS | 62990 | 5666 |
| | 57966 | 8564 |
| | 77083 | 10325 |
| | 48071 | 9698 |

**Table 6.18: Invalid to Invalid transitions with SWAPTIONS taking 8 cores with L2 shared**

The graph of figure 6.18 is also showing the variation of standard results and the results with modified codes for SWAPTIONS program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark.

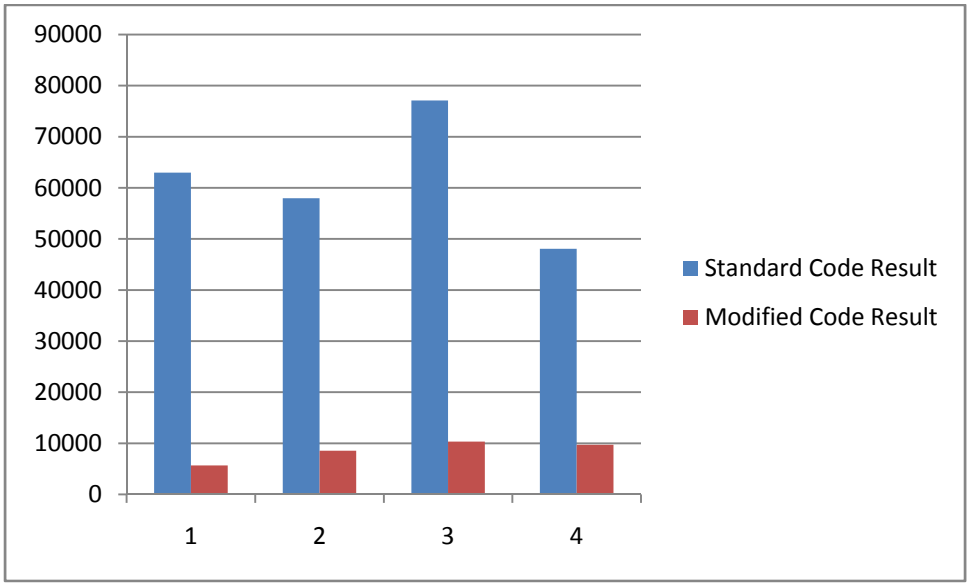



**Figure 6.18: Invalid to Invalid transitions with SWAPTIONS taking 8 cores with L2 shared**

## 6.3 Summary

In this chapter we have introduced the environmental setup which was used while making the experimental results. Finally I have mentioned all the analysis and experimental results that we have got and found that our proposed idea is both time and memory efficient in comparison to standard results.

# Chapter 7: Conclusion and Future work

## 7.1 Conclusions

We successfully implemented a optimized version MESI protocol. The difficulties we faced were that the simulator is very difficult to understand without proper documentation and comments. We spent most of our time in understanding the existing code, compiling, getting results, and implementing and debugging the MESI protocol code. We finally ran PARSEC on our MESI code and compared it to PARSEC run on MESI. We realized that it is very difficult to implement an update based protocol on a simulator that is based on a bus which is not atomic. However, we learnt immensely from this project and thoroughly enjoyed looking at cache and bus logs to find bugs in our code.

The simulation results have shown that we have made invalid to invalid transitions zero for dual cores with CANNEAL, FERRET and SWAPTIONS programs of PARSEC benchmark with level 2 cache as SHARED and PRIVATE.

The simulation result of CANNEAL program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark with level 2 cache as shared and private have shown that we have reduced invalid to invalid transition by 99% with quad cores and octet cores.

The simulation result of FERRET program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark with level 2 cache as shared and private have shown that we have reduced invalid to invalid transition by 99% for quad cores and octet cores.

The simulation result of SWAPTIONS program of PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark with level 2 cache as shared and private have shown that we have reduced invalid to invalid transition by 70% for quad cores and octet cores.

## 7.2 Future Scope

Though MARSS (Micro Architectural and System Simulator) is amazingly flawless open source simulator, but as we all know that nothing is perfect that applies with this simulator too.

As it comes to the future scope of project done by me is huge because till now MARSS (Micro Architectural and System Simulator) is the only simulator which is fully open source. One possible improvement which we can suggest is given below.

As if there are two processors P1 and P2, and right now P1 wants to perform write operation but that shared data block is currently occupied by P2 processor, now P1 has to wait at this moment. Instead of waiting like this we can utilize this time more efficiently by making P1 to release all the previously made update. In this way we can have the recent values of the entire shared data blocks which is modified by processor P1.

# REFERNCES


1) Alan E. Charlesworth. The Sun Fireplane System Interconnect. IEEE Micro, 22(1):36{45,January 2002.
2) Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In Proc. of the 15th Annual Int'l Symp. on Computer Architecture (ISCA'88), pages 280–289, 1988.
3) David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV), pages 224–234, April 1991.
4) Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. ACM Transactions on Computer Systems, 11(4):300–318, November 1993. Earlier version appeared in Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V).
5) Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90), pages 148–159, June 1990.
6) David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In Proc. of the 20th Annual

Int'l Symp. on Computer Architecture(ISCA'93), pages 156–168, May 1993. Also appeared in CMG Transactions,Spring 1994.

7) John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), pages 168–176, February 1990.

8) John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP), pages 152–164, October 1991.

9) Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94), pages 302–313, April 1994.

10) Steven K. Reinhardt. Tempest Interface Specification (Revision 1.2.1). Technical Report 1267, Computer Sciences Department, University of Wisconsin–Madison, February1995.

11) Manoj Kumar, Pooja Arora, "A Survey of Cache Coherence Protocols in Multiprocessors with Shared Memory", ICACSEE'2012

12) David A. Patterson and John L. Hennessy. Computer architecture: a quantitative approach. Morgan Kaufmann, San Francisco, 2007.

13) David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. Parallel computer architecture: a hardware-software approach. Morgan Kaufmann, San Francisco, 1999.

14) J.P. Hayes. Computer Architecture and Organization. McGraw-Hill International Editions, San Diego, CA, USA, 1998.

15) J. Duato, S. Yalamachili, and L. Ni. Interconnection networks: An engineering approach. Morgan Kaufmann, 2003.

16) A. J. Smith. Cache memories. Computing Surveys, 14(3):473–530, 1982.

17) Jim Handy. The cache memory book. Academic Press Professional, San Diego, CA, USA, 1993.

18) Technical documentation of pa-7100lc processor. Hewlett-Packard, http://ftp.parisc-linux.org/docs/chips/PCXL_ers.pdf.

19) Youtao Zhang, Lan Gao, Jun Yang, Xiangyu Zhang, and Rajiv Gupta. Senss: security enhancement to symmetric shared memory multiprocessors. HPCA: 11th International Symposium on High-Performance Computer Architecture, pages 352–362, 2005.

20) X. Qiu and M. Dubois. Moving address translation closer to memory in distributed shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems, 16(7):612–623, 2005.

21) Soo-Cheol Oh, Sang-Hwa Chung, and Hankook Jang. Design and implementation of cc-numa card ii for sci-based pc clustering. IEEE International Conference on Cluster Computing, pages 145–151, 2002.

22) G. Chinya, J. Collins, M. Girkar, H. Jiang, G. Lueh, L. Pearce, X. Tian,H. Wang, P. Wang, and S Yakoushkin. Accelerator exoskeleton. Intel Technology Journal, August 2007.

23) Taeweon Suh, Douglas M. Blough, and Hsien-Hsin S. Lee. Supporting cache coherence in heterogeneous multiprocessor systems. Design, Automation and Test in Europe Conference and Exhibition Volume II, page 21150, 2004.

24) E. Atoofian and A. Baniasadi. A power-aware prediction-based cache coherence protocol for chip multiprocessors. IPDPS ’07: IEEE International Parallel and Distributed Processing Symposium, pages 1–8, March 2007.

25) E. Bolotin, Z. Guz, I. Cidon, R. Ginosar, and A. Kolodny. The power of priority: Noc based distributed cache coherency. NOCS '07: First International Symposium on Networks-on-Chip, pages 117–126, May 2007.

26) F. Baskett, T. Jermoluk, and D. Solomon. The 4d-mp graphics super-workstation: Computing + graphics = 40 mips + 40 mflops and 100,000 lighted polygons per second. COMPCON '88: 33rd IEEE Computer Society International Conference, pages 468–471, February 1988.

27) P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. 13th Annual International Symposium on Computer Architecture, pages 414–423, June 1986.

28) M Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache memories. 11th Annual International Symposium on Computer Architecture, pages 384–354, June.

29) J. Borkenhagen and S. Storino. 4th generation 64-bit powepc-compatible commenrical processor design. Processor Design. IBM Server Group Whitepaper, January 1999.

30) J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded powerpc processor for commercial servers. IBM Journal of Research and Development, 44(6):885–898, November 2000.

31) S. R. Kunkel. Personal communication. April 2000.

32) Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In Proceedings of the 17th Annual International Symposium on

Computer Architecture, pages 148–159, Seattle, Washington, June 1990.

33) David Chaiken, John Kubiatowics, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System, volume 26, pages224–234, Santa Clara, California, April 1991.

34) Richard Simoni and Mark Horowitz. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. In Proceedings of the International Symposium on Shared Memory Multiprocessing, pages 72–81, Tokyo, Japan, April 1991.

35) Mark Heinrich, Vijayaraghavan Soundararajan, John L. Hennessy, and Anoop Gupta. A Quantitatitve Analysis of the Performance and Scalability of Distributed Shared Memory. IEEE Transactions on Computers, 48(2):205–217, February 1999.

36) M.S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," Proc. 11th Annual Int. Symp. on Computer Architecture, pp. 348-354, June 1984.

37) Matt Yourst, “PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator”, Proc. ISPASS 2007.

38) Fabrice Bellard, “QEMU, a fast and portable dynamic translator”, Proc. ATEC 2005

39) Paul Barham, et.al., “Xen and the art of virtualization”, Proc. SOSP 2003

40) Hui Zeng, et.al., “MPTLsim: a simulator for X86 multicore processors”, Proc. DAC 2009.

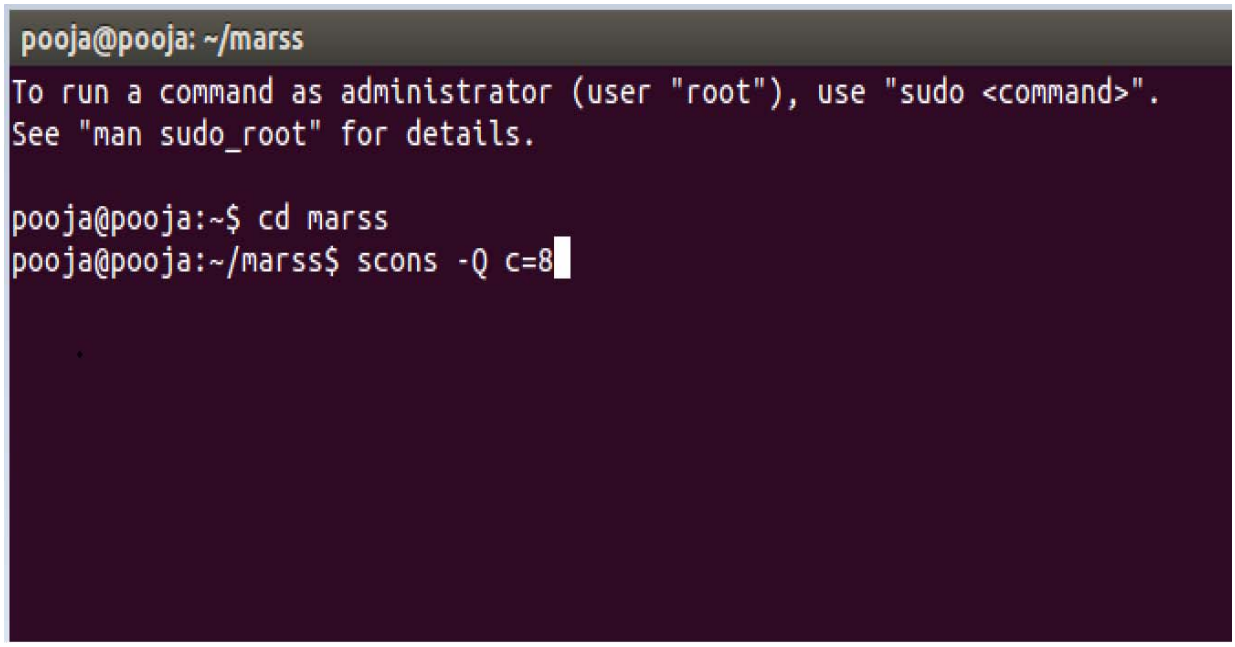
41) Naveen Neelkantam, et.al., “FeS2: Full-System Execution-driven Simulator for x86”, web pages at: http://fes2.cs.uiuc.edu

42) “Zesto: X86 Simulator”, web pages at: http://zesto.cc.gatech.edu

43) “Bochs: IA-32 Emulator”, web pages at: http://bochs.sourceforge.net/

44) Milo M.K. Martin, et.al., “Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset”

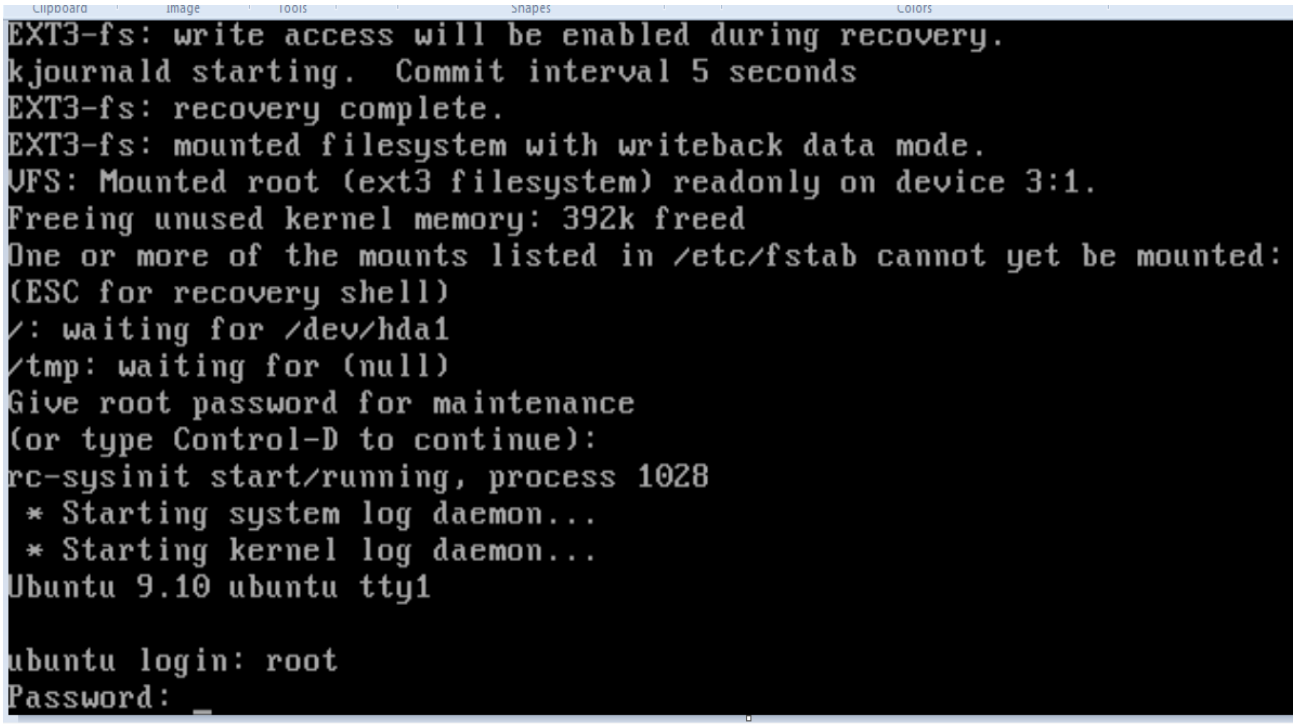
# APPENDIX: SCREEN SHOTS

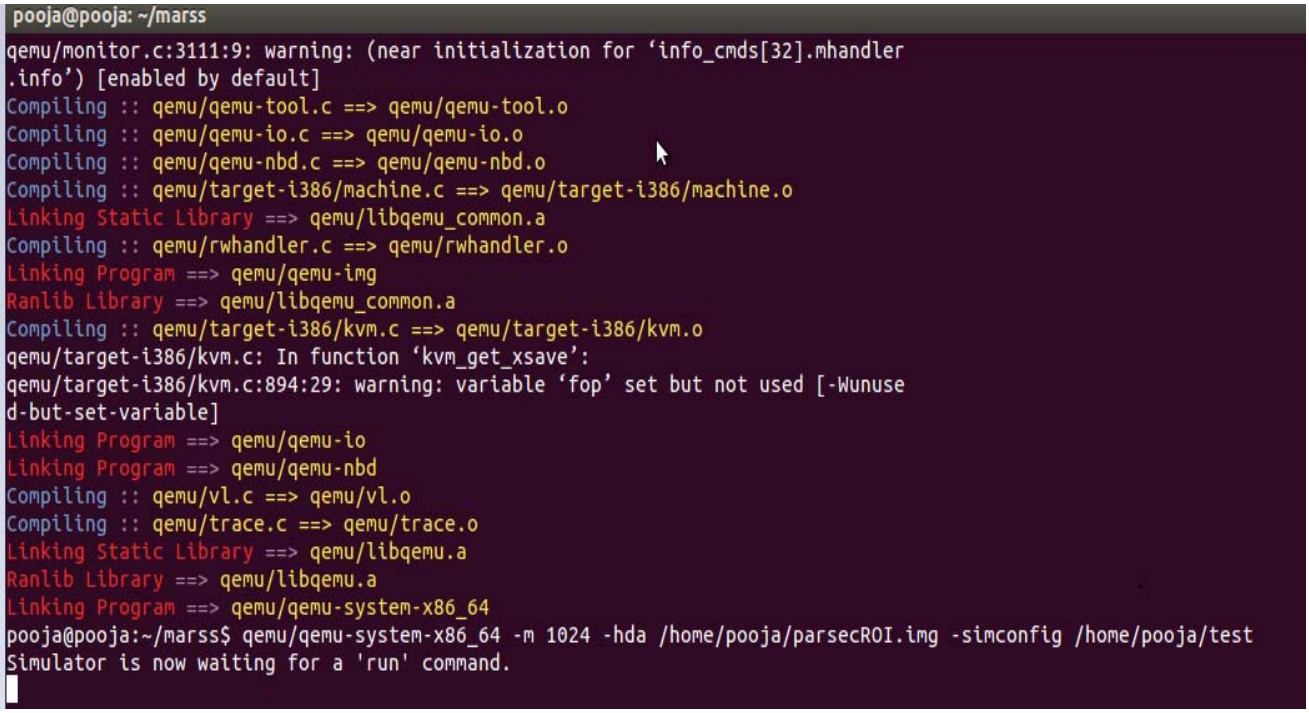



```
Compiling :: qemu/qemu-img.c ==> qemu/qemu-img.o
In file included from qemu/monitor.c:2835:0:
qemu/hmp-commands.h:712:1: warning: initialization from incompatible pointer typ
e [enabled by default]
qemu/hmp-commands.h:712:1: warning: (near initialization for ‘mon_cmds[68].mhand
ler.cmd_new’) [enabled by default]
qemu/monitor.c:3111:9: warning: initialization from incompatible pointer type [e
nabled by default]
qemu/monitor.c:3111:9: warning: (near initialization for ‘info_cmds[32].mhandler
.info’) [enabled by default]
Compiling :: qemu/qemu-tool.c ==> qemu/qemu-tool.o
Compiling :: qemu/qemu-io.c ==> qemu/qemu-io.o
Compiling :: qemu/qemu-nbd.c ==> qemu/qemu-nbd.o
Compiling :: qemu/target-i386/machine.c ==> qemu/target-i386/machine.o
Linking Static Library ==> qemu/libqemu_common.a
Compiling :: qemu/rwhandler.c ==> qemu/rwhandler.o
Linking Program ==> qemu/qemu-img
Ranlib Library ==> qemu/libqemu_common.a
Compiling :: qemu/target-i386/kvm.c ==> qemu/target-i386/kvm.o
qemu/target-i386/kvm.c: In function ‘kvm_get_xsave’:
qemu/target-i386/kvm.c:894:29: warning: variable ‘fop’ set but not used [-Wunuse
d-but-set-variable]
Linking Program ==> qemu/qemu-io
Linking Program ==> qemu/qemu-nbd
Compiling :: qemu/vl.c ==> qemu/vl.o
Compiling :: qemu/trace.c ==> qemu/trace.o
Linking Static Library ==> qemu/libqemu.a
Ranlib Library ==> qemu/libqemu.a
Linking Program ==> qemu/qemu-system-x86_64
pooja@pooja:~/marss$
```

```
Compiling :: qemu/qemu-tool.c ==> qemu/qemu-tool.o
Compiling :: qemu/qemu-io.c ==> qemu/qemu-io.o
Compiling :: qemu/qemu-nbd.c ==> qemu/qemu-nbd.o
Compiling :: qemu/target-i386/machine.c ==> qemu/target-i386/machine.o
Linking Static Library ==> qemu/libqemu_common.a
Compiling :: qemu/rwhandler.c ==> qemu/rwhandler.o
Linking Program ==> qemu/qemu-img
Ranlib Library ==> qemu/libqemu_common.a
Compiling :: qemu/target-i386/kvm.c ==> qemu/target-i386/kvm.o
qemu/target-i386/kvm.c: In function 'kvm_get_xsave':
qemu/target-i386/kvm.c:894:29: warning: variable 'fop' set but not used [-Wunuse
d-but-set-variable]
Linking Program ==> qemu/qemu-io
Linking Program ==> qemu/qemu-nbd
Compiling :: qemu/vl.c ==> qemu/vl.o
Compiling :: qemu/trace.c ==> qemu/trace.o
Linking Static Library ==> qemu/libqemu.a
Ranlib Library ==> qemu/libqemu.a
Linking Program ==> qemu/qemu-system-x86_64
pooja@pooja:~/marss$ qemu/qemu-system-x86_64 -m 1024 -hda /home/pooja/parsecROI.img -simconfig /home/pooja/test
```

```
TCP cubic registered
NET: Registered protocol family 17
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/
nput1
EXT3-fs: INFO: recovery required on readonly filesystem.
EXT3-fs: write access will be enabled during recovery.
kjournald starting.  Commit interval 5 seconds
EXT3-fs: recovery complete.
EXT3-fs: mounted filesystem with writeback data mode.
VFS: Mounted root (ext3 filesystem) readonly on device 3:1.
Freeing unused kernel memory: 392k freed
One or more of the mounts listed in /etc/fstab cannot yet be mounted:
(ESC for recovery shell)
/: waiting for /dev/hda1
/tmp: waiting for (null)
Give root password for maintenance
(or type Control-D to continue): _
```


EXT3-fs: write access will be enabled during recovery.
kjournald starting. Commit interval 5 seconds
EXT3-fs: recovery complete.
EXT3-fs: mounted filesystem with writeback data mode.
VFS: Mounted root (ext3 filesystem) readonly on device 3:1.
Freeing unused kernel memory: 392k freed
One or more of the mounts listed in /etc/fstab cannot yet be mounted:
(ESC for recovery shell)
/: waiting for /dev/hda1
/tmp: waiting for (null)
Give root password for maintenance
(or type Control-D to continue):
rc-sysinit start/running, process 1028
* Starting system log daemon...
* Starting kernel log daemon...
Ubuntu 9.10 ubuntu tty1
ubuntu login: root
Password:


pooja@pooja: ~/marss
qemu/monitor.c:3111:9: warning: (near initialization for 'info_cmds[32].mhandler
.info') [enabled by default]
Compiling :: qemu/qemu-tool.c ==> qemu/qemu-tool.o
Compiling :: qemu/qemu-io.c ==> qemu/qemu-io.o
Compiling :: qemu/qemu-nbd.c ==> qemu/qemu-nbd.o
Compiling :: qemu/target-i386/machine.c ==> qemu/target-i386/machine.o
Linking Static Library ==> qemu/libqemu_common.a
Compiling :: qemu/rwhandler.c ==> qemu/rwhandler.o
Linking Program ==> qemu/qemu-img
Ranlib Library ==> qemu/libqemu_common.a
Compiling :: qemu/target-i386/kvm.c ==> qemu/target-i386/kvm.o
qemu/target-i386/kvm.c: In function 'kvm_get_xsave':
qemu/target-i386/kvm.c:894:29: warning: variable 'fop' set but not used [-Wunuse
d-but-set-variable]
Linking Program ==> qemu/qemu-io
Linking Program ==> qemu/qemu-nbd
Compiling :: qemu/vl.c ==> qemu/vl.o
Compiling :: qemu/trace.c ==> qemu/trace.o
Linking Static Library ==> qemu/libqemu.a
Ranlib Library ==> qemu/libqemu.a
Linking Program ==> qemu/qemu-system-x86_64
pooja@pooja:~/marss$ qemu/qemu-system-x86_64 -m 1024 -hda /home/pooja/parsecROI.img -simconfig /home/pooja/test
Simulator is now waiting for a 'run' command.

```
One or more of the mounts listed in /etc/fstab cannot yet be mounted:
(ESC for recovery shell)
/: waiting for /dev/hda1
/tmp: waiting for (null)
Give root password for maintenance
(or type Control-D to continue):
rc-sysinit start/running, process 1024
 * Starting system log daemon...
 * Starting kernel log daemon...
Ubuntu 9.10 ubuntu tty1

ubuntu login: root
Password:
Last login: Thu Jul  7 18:19:24 EDT 2011 on tty1
Linux ubuntu 2.6.31.4qemu #3 SMP Mon Oct 19 16:41:41 EDT 2009 x86_64

To access official Ubuntu documentation, please visit:
http://help.ubuntu.com/
root@ubuntu:~# ./start_sim _
```

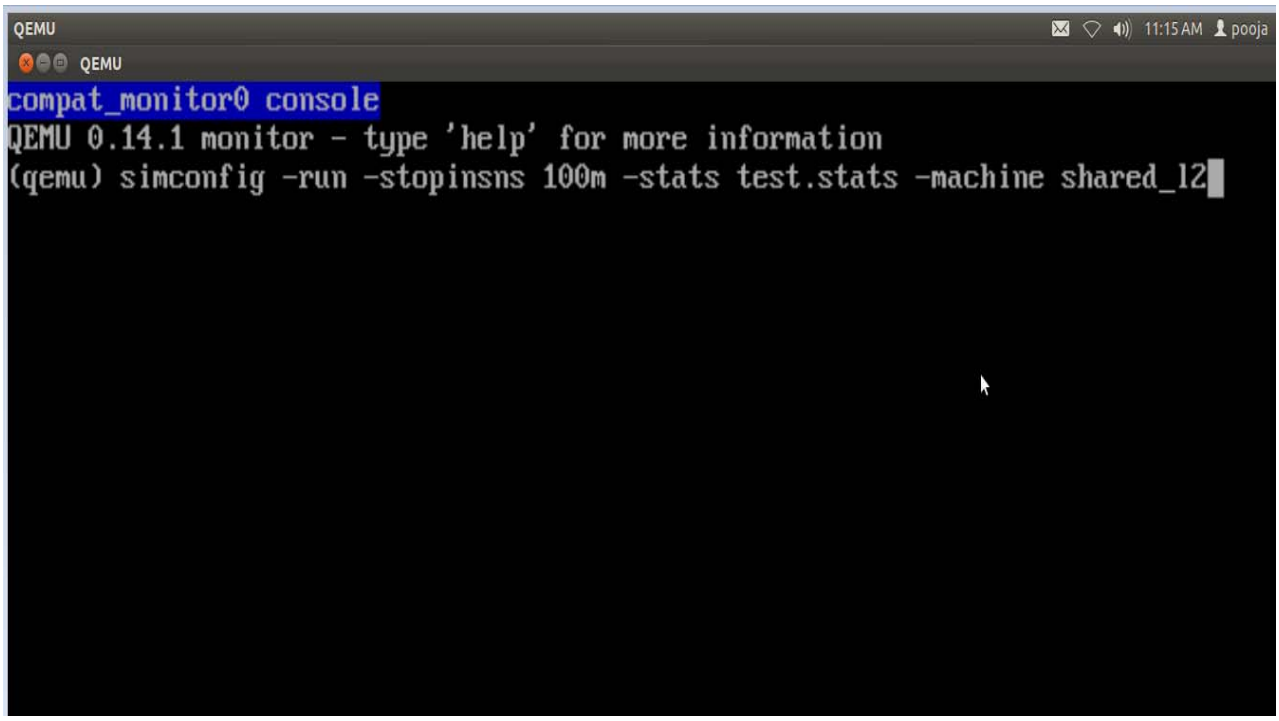
```
pooja@pooja:~/marss$ qemu/qemu-system-x86_64 -m 1024 -hda /home/pooja/parsecROI.img -si
Simulator is now waiting for a 'run' command.
PTLCALL type PTLCALL_ENQUEUE
MARSSx86::Command received : -run
  Completed            0 cycles,            0 commits:         0 Hz,         0
 insns/sec: rip ffffffff81013092 ffffffff81013092 ffffffff81013092 ffffffff81013
  Completed        19000 cycles,        98644 commits:     87780 Hz,    455735
 insns/sec: rip ffffffff81052f85 ffffffff8104f0ff ffffffff8104f80d ffffffff8104f
  Completed        25000 cycles,       230372 commits:     29291 Hz,    643077
 insns/sec: rip ffffffff81052f45 ffffffff8104f0fb ffffffff8104f80d ffffffff8104f
  Completed        31000 cycles,       362089 commits:     29341 Hz,    644137
 insns/sec: rip ffffffff81052f5a ffffffff8104f107 ffffffff8104f805 ffffffff8104f
  Completed        37000 cycles,       493827 commits:     29483 Hz,    647355
 insns/sec: rip ffffffff81052f57 ffffffff8104f107 ffffffff8104f805 ffffffff8104f
  Completed        44000 cycles,       644584 commits:     31116 Hz,    670150
 insns/sec: rip ffffffff81010929 ffffffff8104f0ff ffffffff8104f800 ffffffff8104f
  Completed        78000 cycles,       690113 commits:    161890 Hz,    216785
 insns/sec: rip ffffffff81013091 ffffffff8102bd10 ffffffff8101307a ffffffff81013
  Completed       168000 cycles,       706900 commits:    449859 Hz,     83908
 insns/sec: rip ffffffff81013091 ffffffff81013091 ffffffff81013091 ffffffff81013
```

```
ubuntu login: root
Password:
Last login: Thu Jul  7 18:19:24 EDT 2011 on tty1
Linux ubuntu 2.6.31.4qemu #3 SMP Mon Oct 19 16:41:41 EDT 2009 x86_64

To access official Ubuntu documentation, please visit:
http://help.ubuntu.com/
root@ubuntu:~# ./start_sim
Switching to simulation
ptlsim_ptlcall_init: mapped PTLcall MMIO page at phys 0x8fffff000, virt 0x7f7c
c91000
root@ubuntu:~# cd parsec-2.1/
```

```
Linux ubuntu 2.6.31.4qemu #3 SMP Mon Oct 19 16:41:41 EDT 2009 x86_64

To access official Ubuntu documentation, please visit:
http://help.ubuntu.com/
root@ubuntu:~# ./start_sim
Switching to simulation
ptlsim_ptlcall_init: mapped PTLcall MMIO page at phys 0x8fffff000, virt 0x7f968
19c000
root@ubuntu:~# cd parsec-2.1/
root@ubuntu:~/parsec-2.1# source env.sh
dirname: invalid option -- 'b'
Try `dirname --help' for more information.
dirname: missing operand
Try `dirname --help' for more information.
root@ubuntu:~/parsec-2.1# ./env.sh
root@ubuntu:~/parsec-2.1# ./bin/parsecmgmt -a run -p ferret -c gcc-hooks -x roi
-i simsmall -n 8;
```


QEMU
11:15 AM pooja
QEMU
compat_monitor0 console
QEMU 0.14.1 monitor - type 'help' for more information
(qemu) simconfig -run -stopinsns 100m -stats test.stats -machine shared_l2