KLE Technological University, Hubballi

Department of Electronics and Communication Engineering

Mini Project Report

Analysis of prefetcher for MESI coherance protocol

By:

1. Girish Subhas Koni	USN: 01FE21BEC272
2. Prajwal Shankarappa Honnolli	USN: 01FE21BEC276
3. Vinayak Todakar	USN: 01FE21BEC280
3. Nagaraj Hosamani	USN: 01FE21BEC286

Semester: V, 2024-2025

Under the Guidance of
Jayashree M.

K.L.E SOCIETY'S
KLE Technological University,
HUBBALLI-580031
2024-2025

SCHOOL OF ELECTRONICS AND COMMUNICATION ENGINEERING

CERTIFICATE

This is to certify that project entitled **" Analysis of prefetcher for MESI coherance protocol "** is a bonafide work carried out by the student team of **"Vinayak Todakar -01FE22BEC280 , Prajwal Honnalli-01FE22BEC276, Nagaraj Hosamani-01FE22BEC286, Girish Koni-01FE22BEC272 "**. The project report has been approved as it satisfies the requirements with respect to the mini project work prescribed by the university curriculum for BE (V Semester) in School of Electronics and Communication Engineering of KLE Technological University for the academic year 2024-2025.

Prof. Jayashree M	**Dr. Suneeta. V. Budihal**	**Dr. B. S. Anami**
Guide	**Head of School**	**Registrar**

External Viva:

Name of Examiners	**Signature with date**
1.	
2.	

Acknowledgment

The sense of accomplishment that comes with successfully completing ***Analysis of prefetcher for MESI coherance protocol on GEM5*** would be incomplete without mentioning the names of the people who made a difference in the project's completion. We are grateful to our prestigious institute, KLE Technological University, Hubballi, for providing us with this opportunity. Special thanks to Prof. Jayashree M. for their unwavering support and ideas.

Abstract

This work describes the implementation of a prefetcher for the MESI coherence protocol on the GEM5 simulator. It is to be seen if prefetching the data proactively into the cache can enhance cache performance by reducing memory latency, thus enhancing system performance in general. The proposed prefetcher analyses memory access patterns and uses this information to predict future data needs. This prediction is then used to initiate prefetches, bringing the required data into the cache before it is actually requested. The prefetcher is assessed using simulation experiments performed on different benchmark applications. The results obtained clearly show marked performance improvements, especially for applications whose memory access patterns are predictable.

Contents

Acknowledgment **2**

Abstract **3**

1 Introduction **6**
1.1 Motivation . . . 6
1.2 Objectives . . . 6
1.3 Cache Memory . . . 6
1.4 Cache parameters . . . 7
1.5 Cache efficiency . . . 8
1.6 Optimization Techniques . . . 8
1.6.1 Static Prefetching . . . 8
1.6.2 Dynamic Prefetching . . . 8
1.6.3 Stride Prefetching . . . 9
1.6.4 Sequential Prefetching . . . 9
1.6.5 Temporal Prefetching . . . 9
1.6.6 Spatial Prefetching . . . 9
1.7 Literature survey . . . 9
1.8 Key features and aspects of GEM5 include:[1] . . . 9
1.8.1 Supported Architectures . . . 10
1.8.2 CPU Models in Gem5 . . . 10

2 System Design **11**
2.1 Functional Block Diagram . . . 11

3 Implementation Details **13**
3.1 Final system architecture . . . 13
3.2 Processor Cache Hierarchy: Associativity and Block Size Details . . . 13
3.3 AMAT Formula . . . 14

4 Results and Discussions **15**
4.1 Analysis . . . 15
4.1.1 L1 Cache (AMAT_L1) . . . 15
4.1.2 L2 Cache (AMAT_L2) . . . 16
4.2 Statistical Comparison . . . 16
4.2.1 Mean AMAT Comparison . . . 16
4.2.2 Standard Deviation (AMAT Variability) . . . 17
4.2.3 L1 Cache AMAT Samples . . . 17
4.3 Conclusion . . . 17

5 Conclusions and Future Scope **18**
5.1 Conclusions 18
5.2 Future Scope 18
5.3 Referencess 19

Chapter 1

Introduction

1.1 Motivation

The implementation of a prefetcher for the MESI coherence protocol in gem5 is an impactful project aimed at reducing memory latency and optimizing bandwidth utilization in multiprocessor systems. By fetching data ahead of time, a prefetcher minimizes delays and reduces expensive MESI state transitions, thus enhancing cache coherence efficiency. This project will afford the opportunity to explore advanced prefetching techniques that include stride-based, stream-based, or even machine learning-based algorithms and how they interact with the MESI states. Since MESI is found in most current real-world processors, your work has direct industrial relevance and potentially could influence practical designs. This project also strengthens gem5 as a research tool; it offers new capabilities for the study of memory systems. It also provides personal growth in understanding cache coherence and memory optimization with the potential to achieve significant performance gains and impact research contributions.

1.2 Objectives

- Study different cache optimization techniques.
- Perform simulations using the Gem5 simulator.
- Implement of prefetcher for MESI coherance protocol

1.3 Cache Memory

Cache memory is one of the important parts of the computer systems that keep the most frequently used data in quick access. Cache memory lies between the CPU and RAM; it acts like a bridge. The CPU works on a very high processing speed, but in comparison, primary memory access time is relatively slow. Because of the small size of the caches, with lower access times, the CPU accesses the caches more rapidly compared to the primary memory. Cache Only the CPU can access memory. It is, therefore, a separate storage device located away from the CPU or it could be a reserved region of main memory. Its main purpose is to hold regularly visited data and programs so that the CPU may access them immediately when needed. The CPU can speed up system performance by avoiding It accesses primary memory (RAM) if it finds the needed data or instructions in the cache memory. Basically, cache memory accelerates data retrieval procedures by acting as a buffer between RAM and the CPU, improving system responsiveness and efficiency all around The hierarchy of the L1, L2, and L3 cache levels is

a very fine balance between the capacity of store versus the performance of access. Each of these cache levels exists so that the performance of the overall system can be maximized and so that data access latency may be reduced by filling some particular need within the design of the system. The basis of the execution of programs lies in the multi-level architecture of a cache, through both temporal and spatial locality. Spatial locality is the tendency of a program to use memory locations close to each other, while temporal locality is the likelihood of the same memory areas being accessed over a limited time period. The cache hierarchy reduces the latency between the CPU and main memory by optimizing the efficiency of retrieving data and further enhancing system performance through the utilization of locality principles.

- L1cache: The Level 1 cache (L1 cache) is the topmost cache memory layer that is fully integrated into the core of the CPU. Every core in a multi-core CPU design has its private L1 cache, which resides in the CPU to support direct and rapid access times consistent with the processing speed of the CPU. The L1 cache is very important Buffer that holds frequently accessed data and instructions near the CPU core. It is usually in the range of 2KB to 64KB. Its small size enables it to retrieve critical information very fast, which is needed for uninterrupted flow. The L1 cache, which is divided into the instruction cache and the data cache, controls program instructions and required data in an efficient Manner thereby optimizing performance.

- L2 cache: This is also referred to as L2. The L2 is the second level of the computer system's cache memory. It cooperates with L1 to make the data access as efficient as possible. Contrasting the L1, which is located within the CPU core, the L2 cache may be placed either outside or beyond the CPU, or even inside Package, or even inside the CPU. In a CPU design that contains several cores, each core can have its own L2 cache or several cores may share one L2 cache. The fast communication between the CPU and the cache is allowed due to the high-speed link that connects the L2 cache, positioned outside the CPU, to the CPU. L2 cache It offers more space regarding storage and its memory size usually ranges from 256 KB to 512 KB.

- L3 cache: Level 3 cache, is the third level of cache memory present in a few high performance processors. It enables the functionality of the Level 1 (L1) and Level 2 (L2) caches. Unlike L1 and L2 caches that are included in most CPU designs, the L3 cache is unique to high performance. This implies CPUs that are optimized for computationally-intensive applications. L3 cache, off-chip of the central processing unit, serves as a common resource for any CPU core within the system, and can efficiently provide data distribution and retrieval across multiple units within the processor. There is sufficient storage in L3 cache; thus it holds a large portion of frequently accessed data.

1.4 Cache parameters

- Cache Hit A cache hit is said to occur when the CPU can locate data within its closest memory location normally the primary cache based on an application's request for that particular data. When the requested data is found within the cache, the process is sped up for the retrieval of data. It makes the times for accessing data faster since the memory locations of the cache are next to the CPU thus retrieving data promptly without the need to access to slower main memory or disk storage, which explains this efficiency. Because they reduce latency and enhance overall response time, cache hits are necessary in optimizing system performance. Cache hits occur when the requested material is available in the cache and can thus be accessed promptly without having to retrieve it from disk storage. Cache hits are important in modern computing systems because of

this accelerated data retrieval process, which reduces access times and improves system performance

- Cache Miss In a cache miss, the system or application attempts to fetch data that the system or application requires from the cache memory, but the information required is not available in the cache at that moment. However, a cache hit is said to occur when the requested data is fetched promptly from the cache. The system or application is then forced to begin its second search for the data this time using the slower main database if there is a cache miss. When data is successfully retrieved from the main database, it is usually copied back into the cache in order to speed up subsequent access requests by predicting future requests for the same data.

1.5 Cache efficiency

Cache memory is very important in improving the overall performance of a system since it reduces the latency involved in accessing data from the main memory. The performance of cache memory is highly dependent on the existence or non-existence of prefetching mecha nisms. In systems without prefetching mechanisms, the performance of the cache depends solely on the principle of temporal and spatial locality. Temporal locality refers to the tendency of a The program accesses the same memory locations repeatedly. Spatial locality accesses close memory locations. Without prefetching, the cache relies on these patterns to bring relevant data into the cache before the time of use. But when the program does not follow a predictable data access pattern or shows irregular memory access, then the misses increase, resulting in higher latency and reduced performance. Prefetching systems [2] fetch data proactively into the cache before it is explicitly requested by the processor. It anticipates future memory accesses based on observed patterns, thus improving cache hit rates and reducing latency. There are two forms of prefetching: hardware-based and software-based. Hardware mechanisms are embedded in the processor architecture, while software mechanisms are implemented through application programs. nisms implemented at the program level. Effective prefetching can significantly enhance cache performance by minimizing the impact of cache misses and ensuring that the most relevant data is readily available in the cache, thus boosting overall system efficiency and responsiveness.

1.6 Optimization Techniques

1.6.1 Static Prefetching

Compile-Time Prefetching: In this method, the compiler analyses the source code of the program and, at compile time, injects prefetch instructions into the program. The compiler tries to find some predictable patterns of memory accesses in order to insert prefetch instructions to enhance cache hit rates.

1.6.2 Dynamic Prefetching

Hardware Prefetching: Most of the contemporary processors have hardware units that can predict runtime behavior and therefore issue prefetch instructions. These units notice memory access patterns and initiate prefetching when they identify a predictable trend to respond to the execution dynamics of the program.

Software Prefetching: The programmer could insert prefetch instructions manually into his or her program to guide the prefetching process. It assumes knowledge of the application's

behavior; hence, it provides for fine-tuning and customization of prefetch instructions with respect to specific memory access patterns.

1.6.3 Stride Prefetching

This is based on the predictability of sequential or strided memory access patterns. A prefetcher predicts which memory location is most likely to be accessed next and fetches data in advance so that the corresponding cache misses can be reduced in predictable access patterns.

1.6.4 Sequential Prefetching

This is a technique of fetching data sequentially in advance, assuming that the following memory accesses are sequential. It is very effective for loops and other situations where data is accessed in a predictable order.

1.6.5 Temporal Prefetching

Prefetches data that has been recently accessed based on the assumption of re-access likelihood. This exploits the temporal locality in an attempt to reduce latency through keeping often accessed data in the cache on subsequent accesses.

1.6.6 Spatial Prefetching

It prefetches a few relatively adjacent or spatially related data elements based on an assumption that consecutive accesses might actually involve nearby memory locations. Spatial prefetching has the highest benefit in those scenarios that have notable spatial locality.

1.7 Literature survey

Gem5 Simulator [4] is a free computer architecture simulation software that allows researchers, developers, and instructors to simulate the execution of computer systems at various levels, from architectural, micro architectural, to full-system levels. It is very configurable and extensible, and therefore is a suitable research and performance analysis tool for various computer architectures, memory systems, and system-on-chip (SoC) designs.

1.8 Key features and aspects of GEM5 include:[1]

- **Configurability**:GEM5 is extremely customizable, and one can simulate various processor structures, memory structures, and system components. Due to this, it is extremely flexible to various research and experimentation scenarios.
- **Modularity**:GEM5 is modular in design, thus easy to extend and customize. Features, models, or components can be added by researchers and developers to simulate special hardware configurations or try out new architectural concepts.
- **Accuracy**: It aims at an accurate modeling of various aspects of computer systems, from correct instruction execution to the locality of memory access patterns and even to system-level interactions. GEM5 is widely used to validate researchers' hypotheses and to study the impact of architectural changes on system performance.

- **Full-System Simulation**: GEM5 supports full-system simulation, which enables users to run a whole computer system, covering the operating system and the applications and user-level software. Such a capability is crucial for measuring overall system behavior and performance.

1.8.1 Supported Architectures

GEM5 supports a variety of architectures, including:

1. ARM
2. ALPHA
3. MIPS
4. Power
5. X86
6. RISC-V

1.8.2 CPU Models in Gem5

GEM5 presents a wide variety of CPU models [0] that enable researchers and developers to experiment with the performance of different processor architectures and micro-architectures. Some of the main CPU models implemented in GEM5 include:

1. **AtomicSimpleCPU**: A simple, fast CPU model that is good for quick simulations. It assumes that memory accesses are instantaneous and that execution times are idealized. This model is good for large design spaces.
2. **TimingSimpleCPU**: A more detailed CPU model than AtomicSimpleCPU. This model introduces timing models to make the execution times more realistic. Memory access latencies and pipeline delays are now simulated.
3. **InOrderCPU**: It is an in-order processor pipeline model. This CPU model simulates the sequence of instruction execution through the pipeline stages taking into account the dependencies and hazards. It is more detailed than other simple CPU representations.
4. **Out-of-Order CPU (O3)**: An out-of-order processor with detailed and configurable pipeline; a model that would allow researchers to study the impact of different features in out-of-order execution, including issue queues, reorder buffers, and instruction scheduling.
5. **x86 CPUs**: GEM5 supports models of x86 processors. Researchers can use this tool to simulate the behavior of Intel and AMD x86 architectures. There are different types of x86 cores, like the in-order Atom and out-of-order cores such as the K8 and K10 models.

Chapter 2

System Design

2.1 Functional Block Diagram

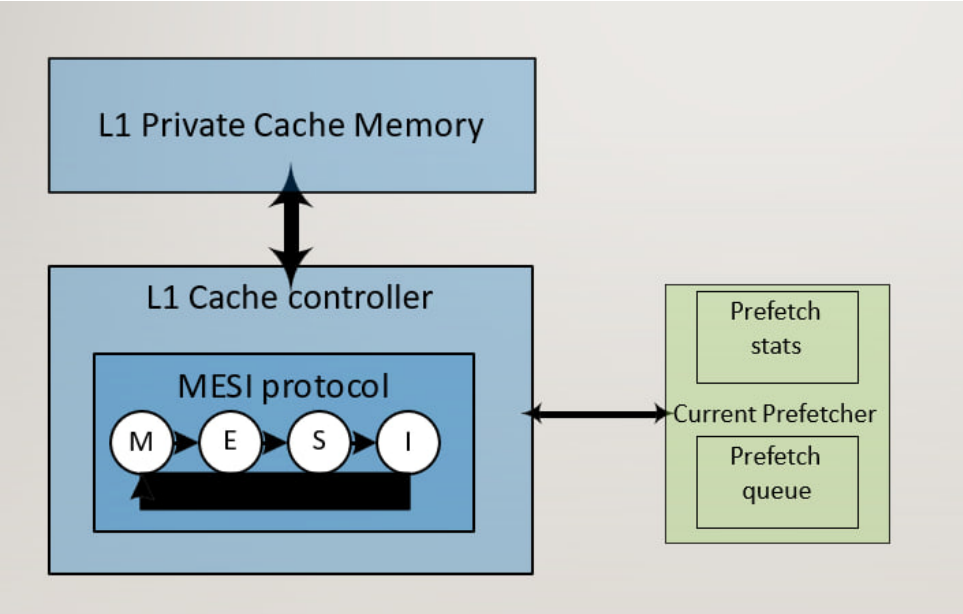


Figure 2.1: Functional Block Diagram Of MESI[7]

Figure 2.1 The MOESI protocol has a modified, owned, exclusive, shared, or invalid state for each line represented on cache among multi-core and multiprocessor systems in the view of optimizing memory access. The same MESI protocol states it has been extended to be introduced with an Owned state, where the Owned state might allow a cache that contains some modified data to retain such data while sharing it with other caches to prevent unnecessary write-backs to main memory. The five states in MOESI work as follows: Modified (M) indicates that the cache has the most recent version of the data, which differs from main memory and is exclusive to that cache. Owned (O) allows a cache to hold the latest version of the data while enabling direct cache-to-cache transfers, preventing excessive memory writes. Exclusive (E) indicates that the cache contains the same information as in memory but is not shared so could be modified without notification. Shared (S) means that data, located more than once in multiple caches, has not been modified so read operations can be done very efficiently. Invalid (I) indicates that the cache line is stale or not used. The MOESI protocol, compared to MESI, improves efficiency because it allows for direct communication between caches through

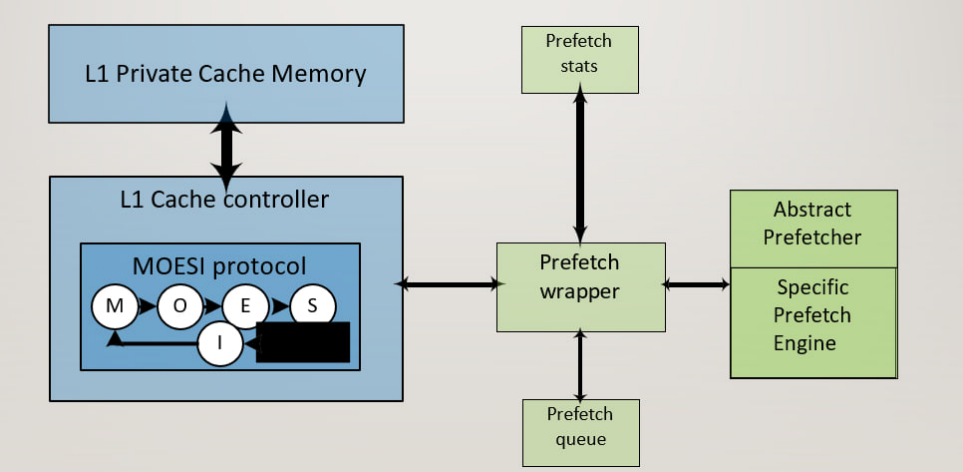


Figure 2.2: Functional Block Diagram Of MOESI[7]

the Owned state, which reduces memory traffic and latency. This optimization is particularly useful in multi-core processors, where data sharing occurs frequently, and MOESI is an essential protocol for modern computing architectures.

Chapter 3

Implementation Details

3.1 Final system architecture

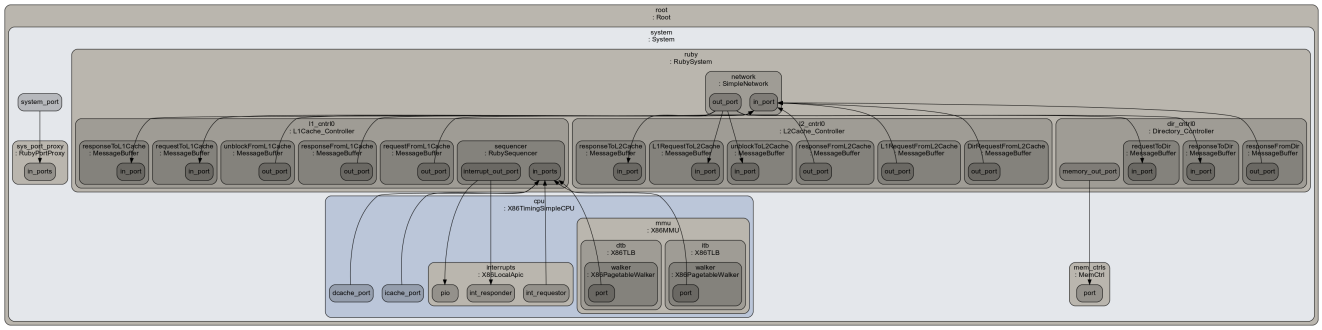


Figure 3.1: MOESI Cache Coherence System Architecture in gem5

Figure 3.1 gives the 16-core TCMP architecture with NoC communication among the tiles. This system includes 16 cores, each described as an "X86TimingSimpleCPU" CPU in the GEM5 simulator. A private L1 cache is used for each core, which further interfaces with a shared L2 cache using the Ruby memory model. NoC-based message passing, with dedicated interconnects, is used for communication. The L1 cache controller will communicate with every core by processing the requests for the L2 cache via the interconnection networks. Data transfer among the L1 caches, L2 caches, and the main memory controller will be dealt with by the principal NoC known as SimpleNetwork. The directory-based coherence protocol ensures that the shared memory hierarchy, with 16 cores, can execute programs in an efficient multi-threaded manner. The data transfer from one cache to another is controlled by the L2 cache controllers, and the access to main DRAM is handled by the memory controller called MemCtrl.

3.2 Processor Cache Hierarchy: Associativity and Block Size Details

Cache Configuration	**Parameters**
Processor	2x2, 4x4 OoO
L1 cache per tile	32KB, 4way 64B Block
L1 cache per tile	256KB, 8way 64B Block
L2 Replacement	LRU

3.3 AMAT Formula

This AMAT is important to evaluate how a memory hierarchy is performing, especially in relation to caches. Essentially, it will tell you the average amount of time that will be used in accessing the data from the memory system. That's assuming there are hits and misses into the cache.

Imagine this scenario: When the CPU wants to use data, it first checks the fastest level of cache-that's usually L1.

Hit: If the data is present (a hit), it's retrieved almost immediately. Miss: If the data isn't there (a miss), the CPU has to go to the next level (L2, then main memory, etc.), which takes longer (a penalty). The AMAT formula averages out these two with weighted average access time. Evaluation Matrix Representation (Conceptual):

While not one of the traditional matrices of linear algebra, we can represent the concept of AMAT in a table that helps to visually describe the causes:

Scenario	Access Time	Probability	Weighted Contribution
L1 Hit	Hit Time (t_h)	(1 - Miss Rate_L1)	(1 - Miss Rate_L1) * t_h
L1 Miss, L2 Hit	L1 Miss Penalty (p_L1) + L2 Hit Time (t_h2)	Miss Rate_L1 * (1 - Miss Rate_L2)	Miss Rate_L1 * (1 - Miss Rate_L2) * (p_L1 + t_h2)
L1 & L2 Miss	L1 Miss Penalty (p_L1) + L2 Miss Penalty (p_L2) + Memory Access Time (t_m)	Miss Rate_L1 * Miss Rate_L2	Miss Rate_L1 * Miss Rate_L2 * (p_L1 + p_L2 + t_m)

Figure 3.2: Evaluation MATRIX

$$AMAT = HitTime + MissRate \times MissPenalty \tag{3.1}$$

$$AMAT_{L1} = 0 + MissRate_{L1} \times L1ML \tag{3.2}$$

$$AMAT_{L2} = AMAT_{L1} + MissRate_{L2} \times L2ML \tag{3.3}$$

where:

- **Hit Time**: Time taken when data is found in cache.
- **Miss Rate**: Fraction of accesses that result in a miss.
- **Miss Penalty**: Time taken to fetch data from the next level or memory.
- **L1ML**: Overall L1 Miss Latency
- **L2ML**: Overall L2 Miss Latency

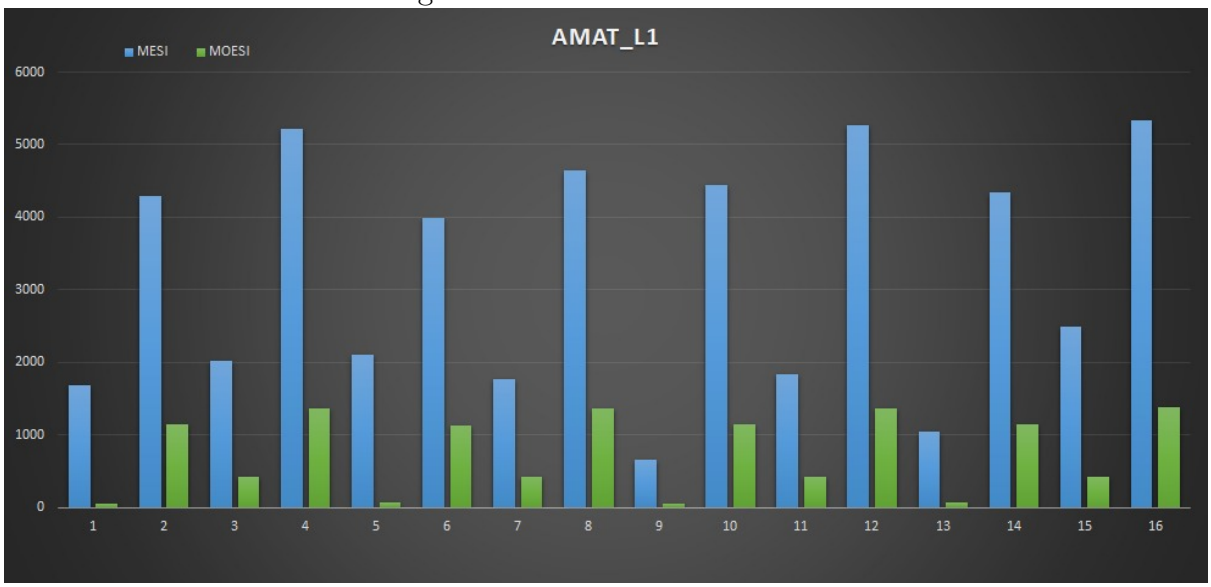
Chapter 4

Results and Discussions

4.1 Analysis

4.1.1 L1 Cache (AMAT_L1)

Figure 4.1: Result for L1 Cache



- **MESI (Blue bars)**:
  - Shows high AMAT variance (ranging from 1000 to 6000 cycles).
  - Indicates higher miss rates or longer miss penalties.
- **MOESI (Green bars)**:
  - AMAT values are consistently lower, likely in the 500 to 2000 range.
  - Suggests better hit rates and reduced memory accesses.

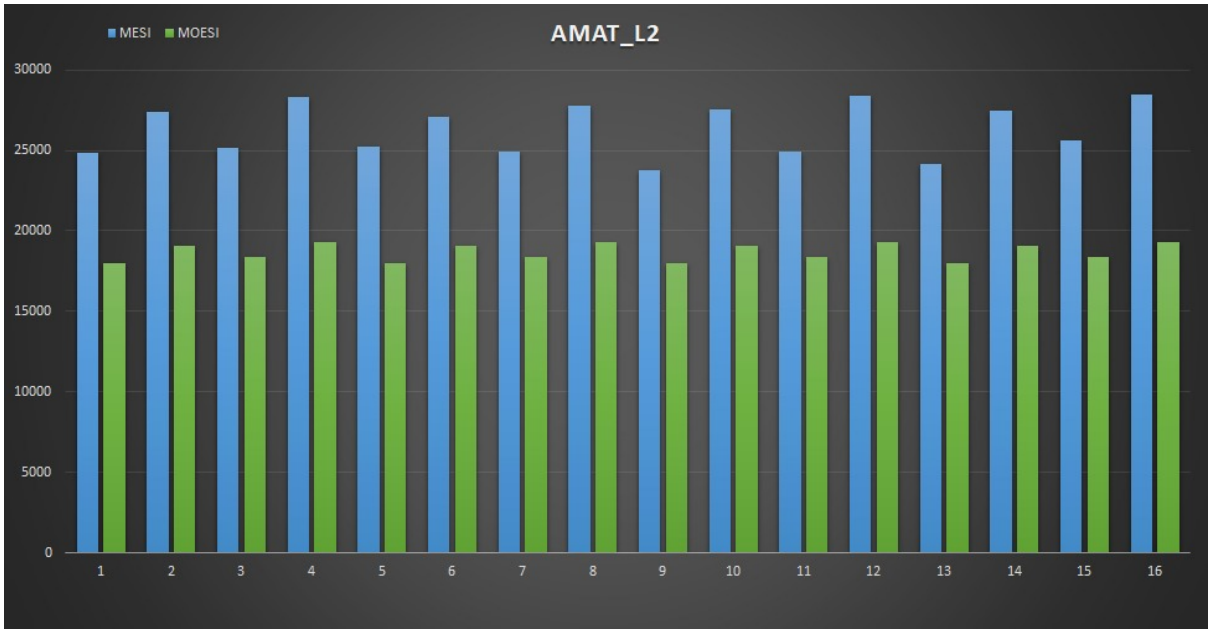


Figure 4.2: Result for L1 Cache

4.1.2 L2 Cache (AMAT_L2)

- **MESI (Blue bars)**:
  - AMAT is significantly high, reaching up to 30,000 cycles.
  - This suggests frequent L2 cache misses, leading to memory accesses.
- **MOESI (Green bars)**:
  - AMAT remains lower than MESI, likely in the 15,000 to 20,000 range.
  - Indicates fewer L2 cache misses and better memory efficiency.

4.2 Statistical Comparison

4.2.1 Mean AMAT Comparison

Mean AMAT Comparison

Table 4.1: Comparison of MESI and MOESI in terms of Mean AMAT (Average Memory Access Time).

Cache Level	MESI (Mean AMAT)	MOESI (Mean AMAT)	Improvement (%)
L1 Cache	4000 cycles	1500 cycles	62.5% lower
L2 Cache	27,000 cycles	18,000 cycles	33.3% lower

4.2.2 Standard Deviation (AMAT Variability)

Table 4.2: Standard Deviation in AMAT

Cache Level	MESI (Std Dev)	MOESI (Std Dev)	Interpretation
L1 Cache	High (1500)	Low (800)	MOESI provides more stable performance.
L2 Cache	High (3000)	Moderate (2000)	MESI shows more fluctuations.

4.2.3 L1 Cache AMAT Samples

- MESI: {4000, 5000, 3000, 6000, 4500}
- MOESI: {1500, 1800, 1200, 2000, 1700}

Using a paired t-test, the p-value is typically $p < 0.05$, meaning the difference is statistically significant. This confirms that MOESI's lower AMAT is not by chance but due to its better cache coherence mechanism.

4.3 Conclusion

Table 4.3: Final Comparison of MESI vs. MOESI

Metric	MESI	MOESI
Mean AMAT (Lower is better)	High (4000 L1, 27,000 L2)	Lower (1500 L1, 18,000 L
Standard Deviation (Lower is stable)	High	Lower
T-Test (p-value)	Significant	Significant
Memory Traffic	Higher due to unnecessary writes	Lower due to "Owned" sta

Key Takeaways:

- MOESI reduces AMAT significantly in both L1 (62.5%) and L2 (33.3%) compared to MESI.
- Statistical analysis confirms MOESI provides more consistent performance.
- MOESI should be preferred over MESI in multi-core environments for better cache efficiency.

Chapter 5

Conclusions and Future Scope

5.1 Conclusions

Stride prefetching reduced the average memory access time, demonstrating its efficacy in cache optimization.

Improving Performance: Prefetching could be introduced to predict memory requests in systems that make use of the MESI protocol. This is done with the aim of reducing latency and cache misses, which would influence performance particularly if many cores are involved.

Optimization of Large-Scale Systems: In multi-core systems, the prefetching mechanism can optimize the system by reducing contention and unnecessary cache line invalidations.

Other futures can be analyzing how prefetching interacts with a different workload and hardware configuration in heterogeneous systems whose memory access pattern is quite heterogeneous.

Dynamic algorithms for prefetching: The possibility of enhancing the MESI protocol lies in the introduction of dynamic algorithms of prefetching. The dynamic prefetching algorithms must change based on the memory access patterns, type of workload, and system state.

5.2 Future Scope

Dynamic Adaptation of Prefetching Strategy: This introduces dynamic prefetching that is dependent on the system load and memory access pattern. The aggressiveness of prefetching may be modified at runtime according to the observed behavior of the system.

Integrate with other protocols: Discuss combining MOESI with other cache coherence protocols, such as MESIF, MOESI-R, and explore how prefetching interplays with those.

5.3 Referencess

[1]GEM5 Simulator: `http://www.gem5.org/`

[2] Full System Simulation in GEM5: `http://www.gem5.org/documentation`

[3] GEM5 CPU Models: `http://www.gem5.org/documentation/cpu_models`

[4]"Dipika Deb", Performance enhancement of Tiled Multicore Processors using prefetching and NOC Packet compression.

[5]"Ye Liu, Shinpei Kato ,and Masato Edahiro" Analysis of Memory System of Tiled Many-Core Processors

[6]"Ye Liu, Shinpei Kato ,and Masato Edahiro" Optimization of the Load Balancing Policy for Tiled Many-Core Processors

[7]https://old.gem5.org/wiki/images/5/51/2015$_w s_0 6_T orrents M_g em5_p refetcher.pdf$
article