Performance study of Last Level Cache in Asymmetric multi core processors

PROJECT REPORT

Submitted by

MURALI DADI

in partial fulfillment of the requirements

for the award of the degree of

MASTER OF TECHNOLOGY

Department Of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

JUNE 2021

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

2021

CERTIFICATE

This is to certify that this thesis (or project report) entitled ***"Performance study of Last Level Cache in Asymmetric multi core processors"*** submitted by **MURALI DADI** to the Indian Institute of Technology Madras, for the award of the degree of **Masters of Technology** is a bona fide record of the research work done by him under my supervision. The contents of this thesis (or project report), in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma..

Dr. T. G. Venkatesh
Research Guide
Associate Professor
Department of Electrical Engineering
IIT Madras 600036

Acknowledgment

First and foremost, I would like to express my deepest gratitude to my guide, **Dr. T G Venkatesh**, Associate Professor, Department of Electrical Engineering, IIT Madras, for providing me an opportunity to work under him. I would like to express my deepest appreciation for his patience, valuable feedbacks, suggestions and motivations.

I convey my sincere gratitude to Shubang Pandey, MS Scholar, IIT Madras and Aparna Behara, PhD Scholar, IIT Madras, for all their suggestions and support during the entire course of the project. Throughout the course of the project they offered immense help and provided valuable suggestions which helped me in completing this project.

I would like to extend my appreciation to all my friends and for their help and support in completing my project successfully.

Abstract

The present workloads and applications are highly diversified, demanding solutions that can deal with critical issues such as the Power Wall, Frequency Wall, and Memory Wall Problem. Asymmetric Multicore Processors (AMP) appear to be a promising approach for addressing these issues in a wide range of applications. Because of the heterogeneity in the AMPs, we can see that the different Cache levels are highly affected by continuous resource allocation and sharing. This heterogeneity in the AMPs arises either due to varying core frequencies or execution order (i.e., In-order or out-of-order executions). ARM big.LITTLE, is an example of such a real-world core. It combines battery-saving and slower CPU cores(LITTLE) with more powerful and power-hungry cores(big). The performance of L2 and Last Level Cache for several flexible cache architectures for various AMP configurations is investigated in this thesis. The study examines parameters such as L2 and L3 miss rates and total On-chip power consumption for several Multithreaded benchmarks from the Parsec and Splash2 benchmark suites with medium inputs. It also investigates the effects of different Cache coherence protocols and block Replacement strategies on these parameters. Our study presents an intermediate cache design for AMPs between two extremes of fully shared L2 and L3 level Caches and fully private L2 and L3 level Cache, achieving desirable power values with optimal cache miss penalties.

Contents

List of Figures

List of Tables

ABBREVIATIONS

LRU	Least Recently Used
MRU	Most Recently Used
SRRIP	Static Re Reference Interval Prediction
DRRIP	Dynamic Re Reference Interval Prediction
Config	Configuration
CPU	Central Processing Unit
DVFS	Dynamic Voltage and Frequency Scaling
LLC	Last Level Cache

Chapter 1

Introduction

The performance of computational units is increasing every year with an increase in the number of chips integrated on the same chip. In the high-performance computing domain, it is highly desirable to have the best computational speed. But the core computational power is increasing faster than the memory system performance, which leads to the Memory-wall problem. Even though computational-bound problems can be solved, memory-bound problems are major issues faced by chip designers because of the poor performance of memory systems. LLC(Last level cache) of a memory hierarchy has a direct impact on the performance. When data is not found in the last level cache, it initiates a transaction to the main memory, which results in both performance and energy penalties. So it is highly recommended to have an efficient memory system for providing high-performance capabilities to the computational units. Such efficiency is mostly depends upon the different cache management policies. Power consumption is also becoming a major issue in current-generation processors. Power-efficient processors are in great demand for various application domains. Many design techniques like ASIC(Application Specific Integrated Circuit) design, Custom design, and multi-core design are adopted to reduce

power consumption. So many power-related simulation tools Wattch, CACT1, McPAT(Multi-core power, area, timing) are also introduced to analyze and optimize the power. McPAT is an integrated power, area, and Timing tool which is used to model the dynamic power. McPAT is also modeled to produce integrated solutions for multi-core processor power.

1.1 Motivation

There has been an immense growth in chip designing for computing systems. The trade-off between performance and power plays a crucial role in this growth. With the aggressive scaling in IC Technology, power density(w/cm^2) is increased due to an increase in the number of transistors per unit area. Even with high-end computers, power consumption is a key consideration. In addition, the objective function(better performance or low power consumption) may also change depending on the requirements and the operating conditions of a device. For example, in mobiles, it is important to optimize energy for battery life during idle periods and to optimize performance during active times. Several processor design approaches, such as multi-core CPUs, are applied to attain great performance and low power consumption. Last level caches are one of the processor's resources that significantly impact system performance and energy usage. As a result, effective handling of last-level caches is required.It's much more difficult to deal with multi-core CPUs which shares the resources like L2 and Last Level caches.

1.2 Aim

In this thesis, we have studied the performance aspects of LLC in both single-core and multi-core architectures. In single-core architecture, we have studied the effect of both LLC size and associativity on the system performance. Primarily, the impact of the different replacement policies over the variable size

and associativity is examined. In multi-core architecture, managing the shared last-level caches is a critical task. Thus we have explored the effect of different configurations for L2 and last level caches which affects the key system metrics such as L2 miss rate, L3 miss rate, and total power consumption of a multi-core architecture. Further in our study, we want to investigate the impact on the power consumption of an Asymmetric multi-core processor due to different cores (with different frequencies) and also different order of execution (In order or out of order).

1.3 Major contribution

As a first step in the thesis, we looked at how size and associativity of Last Level Cache affect its performance. Later we evaluated the L2 miss rate, L3 miss rate, and total on-chip power consumption of several configurations of a multicore architecture with 16 cores by changing the L2, L3 level caches from fully shared among all cores to fully private to each core. Finally, selectively simulated the three different configurations to enhance the study further.

1.4 Outline of the report

The remainder of this thesis is organized as follows. Initially in Chapter 2, a brief background is given related to caches, it's management policies and also about multi core architectures. Then in Chapter 3 we have discussed about the literature survey that are related to our work. Later in Chapter 4, the simulation results for the different configurations is given. Finally in Chapter 5, the conclusion and the future scope related to our study are given.

Chapter 2

Background

This chapter initially reviews the memory-wall problem and then discusses the caches and their different management strategies. Later we also review the multi-core architecture and its types. Finally, we study the parameters that affect the power consumption in an IC(integrated circuit).

2.1 Memory-wall problem

As VLSI technology is improved, we have gone through so many advancements in IC Design. So we can integrate as many computational units on a single chip, thus achieving good computational power. However, even if we reached tremendous speed on computational units, these computations have to be performed on stored memory data. The longer it takes to get data, the slower the computation goes. So the high-performance computation has always been very data-hungry. But even today's fastest storage is slower than the fastest CPU processing speed, resulting in more memory access latency, leading to performance degradation. Here memory bandwidth is limiting the instruction execution rate, which is called a Memory-wall problem. Consider the Fig.2.1

which shows improvement in both memory technology and processor technology.

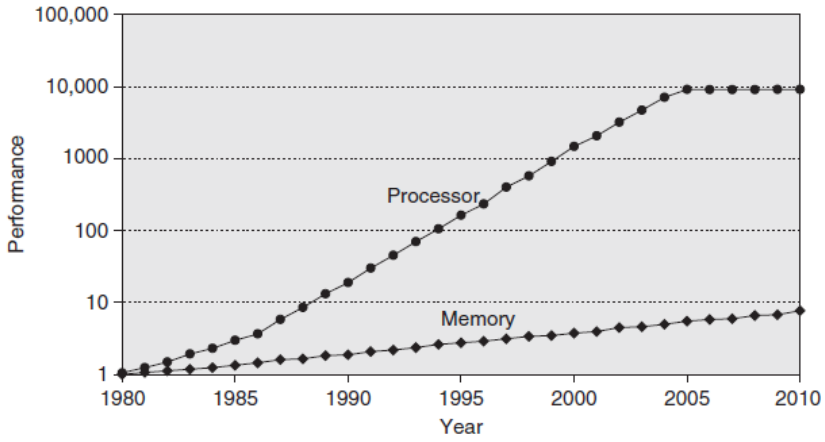


Figure 2.1: Year wise performance improvement in processor and memory technology

Being able to access an unlimited amount of data with unlimited bandwidth is the ideal requirement for better processing. A memory hierarchy is used to bridge the gap between CPU and memory technology, allowing us to access more data at faster speeds.

2.2 Memory Hierarchy

In memory hierarchy, memory is organized into different levels based on their size and access latency, as shown in Fig.2.2. The main aim of the Memory hierarchy is to get more speed using the fastest memory and more capacity from the bigger memory in the hierarchy.

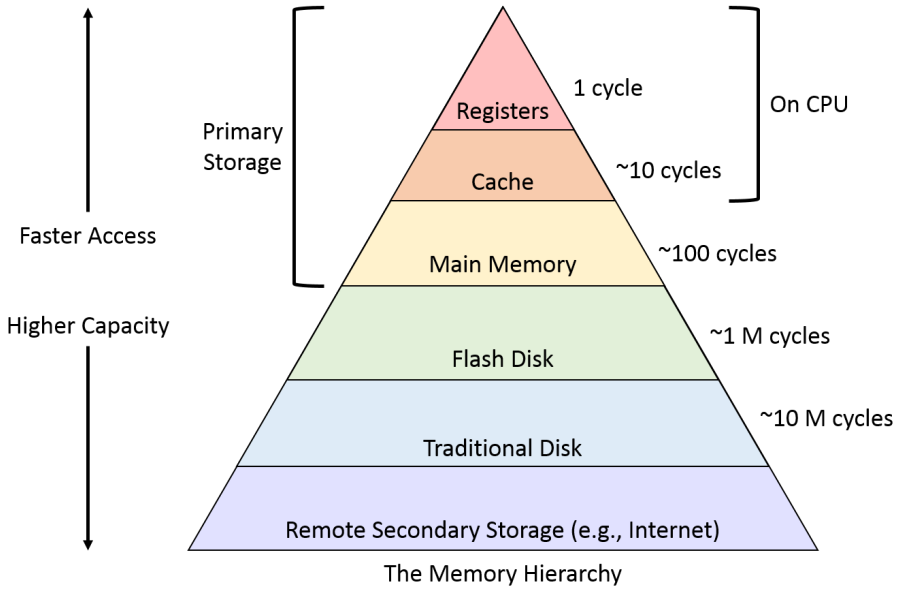


Figure 2.2: The memory hierarchy

The capacity and cost per bit of registers are high where their access latency is very low. But the access time of main memory is in order of ms whose capacity is more. The cache is used to bridge the latency gap between the processor and main memory, whose access time is shorter than that of the main memory and can be up to *Mega Bytes* of size.

Caches are smaller in size but faster in access. And also, We do not want all the instructions or data to be available at the same time. The cache uses the principle of locality of reference to get the data that is going to be accessed many times or in the near future. There are two types of localities, temporal locality, and spatial locality.

1. Temporal locality of reference : This is a locality in time, which means if we have referenced to some address/word, there is a high chance that it is going to be accessed again in the near future. So when CPU is accessing a memory address, if it is not found in the cache, we should bring this from the next level of memory to the highest level of caches so that we

can easily access it the next time.

2. Spatial locality of reference : This is a locality in space. It implies when a memory location is accessed, it is more likely to access additional words in the region around it. When the CPU misses a location, we must also move nearby locations to the highest memory level.

Cache hierarchy has to be designed properly so that it requires less access to the off-chip memory, which takes more time to access data and bring it onto the chip.

2.3 Cache Management polices

Though caches can be accessed quickly than the main memory, because of the limited size of the caches, we should use them properly to have its benefits. So to maintain the most accessible data, proper management policies are required to read, write, and replace, etc.

2.3.1 Write Policies

Data cache is more complicated because we need to read and also write to the data cache. There are two different write policies. They are write-through and write-back.

1. Write-through Cache: When a block is modified, write-through policy simultaneously updates the corresponding block in the main memory so that the main memory and cache are consistent. This is more reliable as it helps in data recovery in case of system failure. But if we have more number of write operations, it always needs to go to the main memory, which degrades the entire system performance.

2. Write-back Cache: The write-back policy prevents the changed cache block from being updated back to the main memory unless it is necessary to do

so. When data of a particular block has to be changed, we only update the cache memory but not in the main memory. Main memory is updated only before the eviction of that particular block. If there are many writes to individual words in a cache block before it is replaced by another, the write-back policy can help performance.

2.3.2 Replacement Policies

Caches are very small in size. We can't place all the things that we need at the same time on the cache. So when new data arrives older data has to be replaced to make space for the new data. We should choose which block to replace(victim block) wisely so that it can't pollute the cache by eliminating the most required block or by placing the unnecessary blocks in the cache. Some of the replacement policies are LRU(Least Recently Used), MRU(Most Recently Used), Round Robin(FIFO), SRRIP(Static Re-Reference Interval Prediction), DRRIP(Dynamic Re-Reference Interval Prediction), Random, etc.

1. LRU : The LRU replacement policy usually removes blocks from the cache that haven't been accessed in a long time. It predicts a near-immediate re-reference interval for all cache lines entered into the cache. This strategy is good for workloads with a greater degree of temporal locality. However, LRU fails to perform well in cases where the reference occurs in the distant future. As a result, in such cases, LRU degrades cache by replacing useful cache blocks.

2. MRU : MRU replaces the most recently used cache block first. It predicts that there is a high chance of accessing the older data in the near future. So it eliminates the latest cache block first. Some researchers have found that for random access patterns and cyclic access patterns, MRU performs better than LRU due to their tendency to retain older data.

3. Random : Implementation of LRU,MRU and other replacement algo-

rithms is cost effective. We can adopt the Random replacement policy to save cost and complexity, but it may come at the expense of performance. The victim of the Random replacement strategy is chosen at random from all the cache lines in the set. It does not require any information about cache line access history. A basic linear feedback shift register is a straightforward technique to accomplish this.

4. Round Robin : It is also called First In First Out. The Round Robin (or FIFO) replacement strategy simply replaces the cache lines in sequential order, starting with the oldest block in the set. The cache evicts the blocks in the order in which they were added, regardless of how often or how many times they have been accessed previously. Every cache memory set comes with a circular counter that refers to the next cache block to be replaced, and the counter is updated every time a cache miss occurs.

2.3.3 Mapping Techniques

There are different mapping techniques to load the data, which is referenced by the *CPU* from main memory to cache memory . They are 1)Direct mapping 2)Set associative mapping 3)Fully associative mapping.

1. Direct mapping : A particular block of main memory can only map to a single line of the cache in the Direct mapping. When new data comes, any data that is currently in a block will be replaced. It takes less hardware as it requires only one tag needs to be checked at a time. There is no need for any special replacement algorithm in direct-mapped caches. Because the main memory block can only map to a single cache line, the new incoming block will always replace the previous block in that line. However, there's a high possibility that two or more frequently used main memory blocks will be mapped to the same cache line, causing useful cache blocks to be replaced. This is called conflict miss. So in the direct-mapped cache, there is a chance for conflict misses.

2. Set-associative mapping : In set-associative mapping, instead of a single cache line, we have a given set of lines where the main memory block can be mapped to. In general, we have 2,4,8,16 ways of set-associative mapping. We need to search a given set of tag bits(2,4,8 or 16) for a particular block. So it takes more hardware and power compared to the direct mapping. But due to more possible cache lines for each memory block, conflict misses are reduced.

3. Fully associative mapping : The fully associative cache can be treated as a single cache set with multiple cache lines. It allows us to place a memory block in any of the cache lines, allowing us to make full use of the entire cache. The fully associative cache can have a much better hit rate than the remaining mapping techniques. But if we want to search for a particular block in the cache, we need to search all the cache lines since it can be anywhere in the cache. So it requires much hardware to compare all the tag bits, which consumes more power. And it also takes more time to iterate through all the cache lines.

2.3.4 Cache Misses

When the address from which the processor wants to read or write is not present in the particular level of cache, it is termed as a cache miss. Due to cache miss, the processor has to wait for a longer time for data as it has to be fetched from the next level of cache or the main memory. This time is termed as a miss penalty. Different types of misses are

1. Compulsory Miss : Compulsory misses are known as cold start misses, or the first reference misses. Every program or application always starts with empty caches unless proper prefetching techniques are used. As a result, when the CPU looks for a block in a cache, it misses, which is referred to as a compulsory miss. These misses occur even in an infinite size cache. Compulsory misses can not be avoided completely.

2. Capacity Miss : When the program working set is substantially greater than the cache capacity, capacity misses occur. The cache cannot contain all the blocks required for a program. When a cache block is replaced due to shortage of space and is accessed again in the future, it leads to capacity miss. Since capacity miss occurs due to limited size, it can be reduced by increasing the cache size. But when cache capacity is more, it requires more area and takes more access time.

3. Conflict Miss : It is also called collision miss. When multiple blocks of main memory are mapped to the same cache set, conflict misses occur. When a cache block is replaced due to conflict, and that block is requested again in the future, it results in conflict miss. More conflict misses occur in the direct-mapped cache due to a single possible cache line for each main memory block, whereas set-associative mapping can reduce conflict misses. If we increase the degree of the associativity of cache, conflict misses reduces. Fully associative caches do not have conflict misses due to the flexibility of placing. But if associativity increases, power consumption increases. Greater associativity also increases the hit time.

2.3.5 Memory access patterns

The order in which the CPU accesses the main memory is known as memory access patterns. Common memory access patterns are

1. Recency- friendly access pattern :

$$(x_1, x_2, ..., x_{k-1}, x_k, x_k, x_{k-1}, ..., x_2, x_1)^N, \ for\ any\ k \tag{2.1}$$

It's a typical N-times-repeating stack access pattern. The re-reference in-

terval for a recency-friendly access pattern is near-immediate. Thus, the access pattern benefits from the LRU replacement policy for any value of K. Other than LRU, any replacement strategy can worsen the performance of certain access patterns.

2. Thrashing access patterns :

$$(x_1, x_2, ..., x_k)^N \tag{2.2}$$

It is accessing a data of length K in a cyclic manner that repeats N times. The working set fits inside the cache when K is smaller than or equal to the number of cache blocks in the cache. But if length of the data exceeds the number of blocks, LRU receives no hits due to cache thrashing.

3. Streaming access patterns :

$$(x_1, x_2, x_3, x_4, ...x_k), k = infinity \tag{2.3}$$

The access pattern has no locality in its references when the length of the data is sufficiently large (i.e., K=infinity). Workloads with an infinite re-reference interval can be considered as streaming access patterns. As a result, under any replacement strategy, this sort of pattern receives no cache hits.

4. Mixed access patterns : A mixed access pattern can be treated as workloads with some references having a short re-reference interval and some other references with distant re-reference interval.

2.4 Multi core Architecture

The basic need of a multi-core processor is parallel computing. Time taken is the main drawback of a single-core processor. So instead of having a single su-

perscalar processor, it is better to have more cores(processing units) on the same chip. Since we have more cores, we can allocate different applications to different cores, which allows better parallel processing and then increases the overall system's performance. Multi-core processors are energy efficient. Because these processors can divide whole work into different cores, which decreases the load on each core compared to single-core processors. Now it is easier to design multi-core processors where each core can perform limited operations than designing a big superscalar processor which can perform all the operations. Though we can achieve high parallel computing from multi-core processors, the size of the performance increase depends on the number of cores, the level of real concurrency in the given application, and the use of shared resources. Based on the type of each core in a multi-core architecture, they are categorized into symmetric and asymmetric multi-core architectures.

2.4.1 Symmetric Multi core Architecture

Homogeneous cores are also called symmetric cores, where all cores are of the same type. This means they all have the same type of processing powers, run at the same frequency, and have the same type of resources like cache size, associativity, block size, etc.

2.4.2 Asymmetric Multi core architecture

Asymmetric multi-core architectures are also called Heterogeneous multi-core architectures. Individual cores with varied execution frequencies, cache sizes, instruction window widths, and other fundamental features(like in-order or out-of-order) result in heterogeneity among multi-cores. The main reason for heterogeneous cores is that this architecture can match different types of applications or phrases in one application to the best-suited core to meet performance demands. While some applications benefit from the most advanced processors, others often under-utilize that hardware and suffer performance loss when run

on little aggressive processors. So when we have heterogeneous cores where some cores with higher capability and other cores with less capability, we can assign applications to cores accordingly, which can efficiently use the entire system.

2.5 Cache Coherence

When we have more cores, each core needs to have some resources like cache memory, etc. Based on how cores accessing cache memory, caches are divided into shared caches and private caches.

2.5.1 Private caches and Shared caches

The private cache is where each core has its own cache memory. Since it is private to each core, the small size of the cache can accommodate sufficient data, which gives less access latency. In the case of private caches, there is a chance of having replicas of the same data. This leads to the under-utilization of the total net capacity of cache and also reduces the hit rate. Private caches can have lower bandwidth interconnects. Private caches are advantageous when there is little data sharing among the cores. When useful cache blocks of a certain core are evicted due to conflicting references made by other cores sharing the cache, it is called destructive interference, resulting in cache pollution. However, because only one core uses the cache, such interference has no effect on performance when using a private cache.

When all cores have one common cache memory, it is called shared cache. When we have a shared cache, it must have a bigger cache to avoid capacity misses. Access latency will be increased due to this bigger cache size. It can efficiently allocate cache among all the cores by providing a sufficient size to the required core, giving a better hit rate. Because shared caches must handle requests from several cores, they require greater bandwidth interconnects. When there is a large degree of data sharing across cores, a shared cache design is beneficial. In the case of a shared cache, memory accesses generated by all

cores may cause constructive interference, in which the other cores utilize one core's prefetched cache line.

2.5.2 Coherence

When each core of a multi-core processing system has its own cache and shares another memory system (such as the last level cache, main memory, and so on), there is a potential that multiple copies of shared data are generated. Each processor that requested the same block has one copy in its local cache and another copy in the main memory. When data in one of these blocks is updated, the other copies must update as well. Cache coherence is the practice of ensuring that changes in shared data values are propagated to all copies in the system. Cache coherence intends that two cores must never see different values for the same cache block.

We need to maintain the consistency of data in all locations. There are two hardware mechanisms to ensure the coherence of shared data in multi-core processors. They are directory-based protocols and Bus based(snoopy) protocols.

I. Bus-based protocols

This is also called snooping. Snooping is a process where each processor observes all other processor's read-write requests and keeps the cache blocks coherent. It changes the cache block state based on the observed actions by a processor on the bus. There are two kinds of snooping protocols based on managing a local copy of write operation..

1. Write-invalidate : In a write-invalidate protocol, when a processor writes to any cached copy, it forces other processors to discard or invalidate their copies. When a processor writes on a cache block whose copy is also there in other caches, all other copies of this block are invalidated through bus snooping. In this case, since the blocks in the other caches are invalidated,

those caches miss that particular block in further accesses. These types of misses are called coherent misses. This category includes protocols such as MSI, MESI, MOESI, and MESIF.

2. Write-update : When a processor writes to a shared cache block in the write-update protocol, all other copies must be updated to reflect the change. This technique sends write data to all caches on a bus. This protocol must search for other copies as well as update those blocks, resulting in more bus traffic than the write-invalidate protocol. Dragon and Firefly are examples of the write-update protocol.

II. Directory based protocols

. This type of protocol uses a directory that keeps track of where the copies of each block reside. Caches consult this directory before accessing a block to ensure coherence. If we want to read or write to a block, we need to make an explicit request to the directory that tells which other caches have that particular block. In this case, we don't have a single point of serialization for all memory requests. Whenever we send a memory request, it is not visible to others in the system like in bus-based protocol. Though the latency of the directory is longer, it uses less bandwidth since messages are point to point and not broadcast. For this reason, systems with more cores use this type of coherence protocol. The scalability of snoopy methods is poor due to the use of broadcasting where directories can be scalable easily.

2.6 Power

In the earlier stages of VLSI, people were concerned about performance, where power consumption is secondary. But when technology scaling down power consumption has also become the most critical issue. This is because of increasing transistor count per unit area. Leakage power was also relatively low up to the

$100nm$ technology, but it has grown significantly with the $32nm$ technology.

2.6.1 Why Low-power design?

First of all packaging and cooling cost of a chip depends upon the amount of power that chip is consuming. More power consumption leads to more cost. Due to the integration of more and more transistors on a single small chip area, power density(w/cm2) is increasing. Mobile phones, laptops, and other battery-operated devices are in higher demand. Furthermore, the functionality of these devices is increasing, demanding greater power. As all these devices are battery-operated, battery life has become a primary concern. When power consumption is increasing, the temperature of the chip increases more unless we provide a better cooling mechanism. It is found that every 10-degree rise in temperature roughly doubles the failure rate. . So to make the chip reliable, we need to have low power consumption. As a result, low power design has become an important aspect of VLSI design, in addition to performance.

There are various types of power consumption in the CMOS-based VLSI chips. They are Dynamic power, short circuit power, leakage power. The power dissipated when a chip is in the active condition like changing the inputs or presence of clock signal etc. is called Dynamic power. When inputs are applied, a lot of charging and discharging occurs at output nodes that consume power. Dynamic power consumed in a circuit is given by:

$$P_{Dynamic} = \alpha * V_{DD}^2 * f_{CLK} * C_L \tag{2.4}$$

where V_{DD} is the supply voltage, f_{CLK} is the frequency of clock, C_L- is the effective load capacitance and α is the switching activity.

So if the clock frequency is increased for better performance, power consumption also increases. This is one of the main key factors behind the growth of multi-core processors to achieve performance rather than the single-core with increased clock frequency which consumes more power.

Chapter 3

Literature Review

In this chapter we review the existing literature works related to the last level cache performance and energy consumption.

In this chapter we review the existing literature works related to the last level cache performance and energy consumption. Cache memories are often employed in microprocessors to increase the system performance, and thus these caches have been the subject of numerous studies. LLC replacement policy is one of the components of modern processors that significantly affects the off-chip miss traffic and power consumption. Vakil-Ghahani *et al.* have suggested a new replacement policy [1] by taking advantage of the correlation between reciprocal of hit counts and reuse distance of a block. This policy makes good replacement decisions based on the remaining number of cache block hits. Peneau *et al.* have studied how different last-level cache replacement strategies affect the system performance, and energy consumption [2].

The asymmetric multi-core architectures are in high demand, and the existing replacement policies have significant challenges when implemented in Asymmetric multi-core systems. Ramtake *et al.* have studied the effect of Associativity on L1, L2 caches in a multi-core system concerning the cache hit ratio and

IPC(Instructions per Cycle) [3]. The latest VLSI chips integrate larger caches on the chip, and managing such larger cache sizes has considerable overhead. Jang *et al.* [4] have suggested a cache design for larger last-level caches, so that good performance is attained even with high granularity. Anandkumar *et al.* have proposed a new hybrid cache replacement strategy for heterogeneous multi-cores that combines LRU, and LFU replacement policies [5]. Heterogeneity in a multi-core system can be achieved by changing individual core frequencies, cache sizes, and other cache parameters. *et al.* have investigated the benefits of having various cache sizes in HMPs(Heterogeneous multi-core processors) and how a scheduling technique can explore such benefits to reduce the overall miss rate [6].

The last-level cache (LLC) in a modern chip-multiprocessor (CMP) is typically shared by all the cores. Processors use the shared caches more frequently; therefore, eviction of the shared data causes more cache misses. Thus, to efficiently utilize the shared LLC on a CMP, Sato *et al.* have proposed cache partitioning to protect the shared data by reducing unnecessary evictions [7] . This approach separates shared and private data and uses cache partitioning to give each type of data its own cache space. Several research works have been done on the partitioning of shared LLC to improve system performance. But they all miss the heterogeneity in the spatial locality of different applications. Gupta *et al.* [8] showed how leveraging spatial locality allows significantly more effective cache sharing. They have highlighted that when large block size is used, the cache capacity requirements of many memory-intensive applications can be dramatically lowered, allowing them to give more capacity to other workloads effectively. In CMPs(Chip Multi Processors), private LLC provides a better access latency than shared caches. But more private caches result in replication of shared data, leading to underutilization of total net capacity of cache, thus decreasing the overall hit ratio. To handle the above mentioned problem, Yuan *et al.* have proposed a new cache management technique that improves the performance of a CMP using the private Last Level Cache [9].

When we integrate hundreds or thousands of cores on a single chip, it be-

comes very complex and expensive to have an efficient cache coherence protocol that maintains the consistency of shared data. Kaur *et al.* have presented a cache coherency management technique for the non-coherent cache architectures that uses an automatic parallelizing compiler [10]. The traditional Snoopy-based coherence protocols are not scalable, and they need much higher bandwidth. The performance of the shared memory multiprocessors is further increased by using an efficient method for addressing the cache coherency. Asaduzzaman *et al.* have offered a hybrid cache coherence protocol (using snoopy and directory techniques) with an improved sharer group mechanism [11]. They have used a directory to check the memory requests, and then the system is updated. Energy consumption has become one of the most critical issues in computer architecture. Although caches can significantly boost system performance, they use a significant amount of overall system power. Chakraborty *et al.* [12] have analyzed the effect of LLC on the chip temperature, and they have proposed a new policy that resizes on-chip LLC at run time so that the leakage power consumption is reduced. To further reduce the energy dissipation in a cache, Zhange *et al.* have proposed a dynamic cache configuration that can tune the degree of the associativity of a cache between one(direct-mapped), two, and four called Way concatenations [13]. When establishing cache coherence, the most important factors to consider are performance, energy, and scalability. Joshi *et al.* have investigated the energy requirement and performance of different snoopy-based and directory-based cache coherence protocols [14].

Chapter 4

Performance Study of Last Level Cache

In this chapter, we have broadly categorized our study into two folds. One-fold is to study the variation of both associativity and cache size with different replacement policies. The other-fold is to check the performance trade-off in heterogeneous Multi-core architecture.

4.1 Study of both associativity and cache size with different replacement policies

This section will study the variation of associativity and cache size with different replacement policies such as LRU, DRRIP, SRRIP, and SHiP. The LRU (Least Recently Used) policy removes blocks from the cache that haven't been accessed in a long time. It consistently anticipates that requested blocks will be re-referenced in the near future. However, applications whose reference is only

made in the distant future perform poorly under LRU. This policy works better for a recency-friendly access pattern.When we need to deal with a larger working set (thrashing) or having bursts of references to non-temporal data(scan), we should go for cache replacement using RRIP(Re-Reference Interval Prediction). There are two policies based on the RRIP: SRRIP (Static Re-Reference Interval Prediction) and DRRIP(Dynamic Re-Reference Interval Prediction). The re-reference interval of blocks is statically predicted by SRRIP to avoid polluting the cache with blocks of distance reuse intervals. For this, SRRIP assigns a Re-Reference Prediction Value(RRPV) to each cache block. N bits can represent RRPV values from 0 to $2^N - 1$ can. Low RRPV value represents near-immediate re-reference. RRPV of $2^N - 2$ is used to represent long re-reference, and RRPV of $2^N - 1$ represents distant re-reference. Because predicting a near-immediate or distant re-reference at cache insertion time isn't always reliable, the SRRIP policy always inserts a new block with a long re-reference interval. RRPV is set to 0 when it is accessed. Hit promotion(HP) and Frequency promotion(FP) algorithms are used to update the RRPV of each cache block further. At the time of eviction, SRRIP looks for a block with RRPV of $2^N - 1$ and replaces it.

SRRIP utilizes the cache inefficiently when the available cache size is smaller than the re-reference interval of all blocks. It leads to cache thrashing and no cache hits in such situations. DRRIP is used to prevent this situation as an extension of SRRIP. We employ Bimodal RRIP (BRRIP) to prevent cache thrashing, which inserts most cache blocks with a distant RRPV and rarely inserts new cache blocks with a long RRPV. However, for non-thrashing access patterns, always using BRRIP can degrade the cache performance. So DRRIP incorporates set dueling monitors (SDM) to determine which replacement policy (SRRIP or BRRIP) is most appropriate for the application. By permanently allocating a few sets of cache lines, an SDM estimates the misses for each given policy (SRRIP and BRRIP). Then, it selects the winning policy using a single policy selection counter. Finally, the DRRIP policy takes the winning policy from the two SDMs and applies it to the remaining cache sets. But in the case of DRRIP also, the learning phase is done after insertion of the block. The

SHiP (Signature-based Hit Predictor) policy addresses this issue by making predictions based on the access patterns of each block. SHiP policy aims to predict whether the insertions by a given signature will receive future hits. If cache insertions by a particular signature are re-referenced, it is predicted that future cache insertions by that signature will also be re-referenced. In addition, if cache insertions by a particular signature do not get subsequent hits, further insertions by the same signature will not receive any hits. To achieve this, SHiP uses signature tables based on the memory region references and Program counter(PC) references. SHiP policy aims to predict whether the insertions by a given signature will receive future hits. It predicts that if cache insertions by a given signature are re-referenced, then future cache insertions by the same signature will again re-referenced. Also, if cache insertions by a given signature do not receive subsequent hits, then future insertions by the same signature will again not receive any subsequent hits.

Simulation Setup used: We have used the ChampSim simulator for this study. The ChampSim is a trace-based simulator that is used in the *Second* Data Prefetching Championship and recently used in the *Second* Cache Replacement Championship. It is primarily used to design simple multi-core Out-of-order processors. The typical traces that we use for our simulation are from SPEC CPU 2006 benchmark suit. The configuration that we used is described in Table 4.1.

Simulation Results:

We broadly categorise the our simulation results as follows:

1. Varrying Associativity of cache
2. Varying LLC cache size

4.1.1 Varying the Associativity of cache

In this section, we fixed the configuration parameters as shown in Table.4.1We wanted to study the effect of varying associativity of last level cache over the

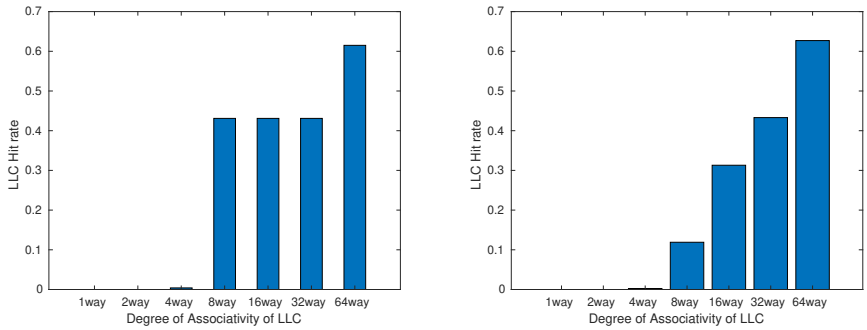
Parameters	Configuration
Number of Cores	1
Benchmark	SPEC CPU 2006-gcc
Inclusion Policy	Non-Inclusive
Block Size	64 Bytes
Order of Execution	Out of Order
$L1$ size	$32KB(I)$, $32KB(D)$
$L1$ Associtivity	$8(D)$, $8(I)$
$L2$ Size	$256KB$
$L2$ Associtivity	8
$L3$ Size	$512KB$
$L3$ Associtivity	16
Off-Chip DRAM size	$4096MB$

Table 4.1: Configuration details - 1

different replacement policies, and the corresponding plots are given in Fig. 4.1. We can observe that the hit rate is zero in the case of $1, 2, 4$ ways of associativity. Because here, we have used synthetic workloads which are designed to test the worst-case performance. Synthetic workloads have very little temporal and spatial locality, which leads to more miss rates. After 32 ways of associativity, all policies are having almost the same hit rate. We know that when the associativity of a cache increases, the hit rate also increases due to a reduction in the conflict misses. So here in the plots, we can observe that the increase in associativity is leading to a better hit rate. Among all the replacement policies, LRU and DRRIP replacement policies give better performance even for the lower associativity with the given workload. For the low degree of associativity(up to 32 ways), SHiP is giving a very poor performance. Wherein in the case of 64 ways of associativity, it is having better performance than the remaining policies.

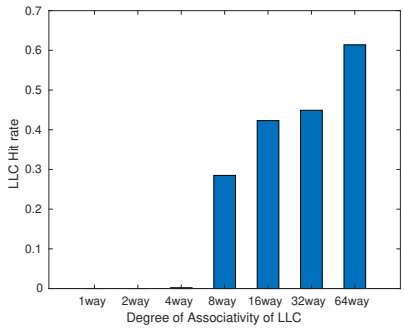
4.1.2 Varying LLC cache size

In this section, we have changed the size of the last level cache by fixing the remaining parameters as shown in Table 4.1. Corresponding simulation results are shown in Fig.4.2. In all replacement strategies, there were no hits until the last level cache size of 512KB, as seen in this diagram. LRU replacement

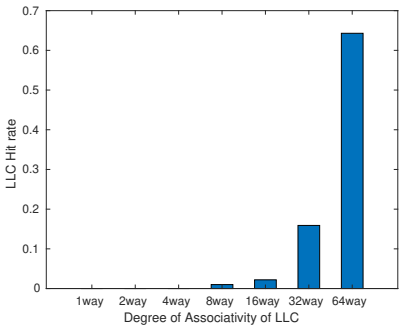


(a) LRU Replacement policy

(b) SRRIP Replacement policy



(c) DRRIP Replacement policy



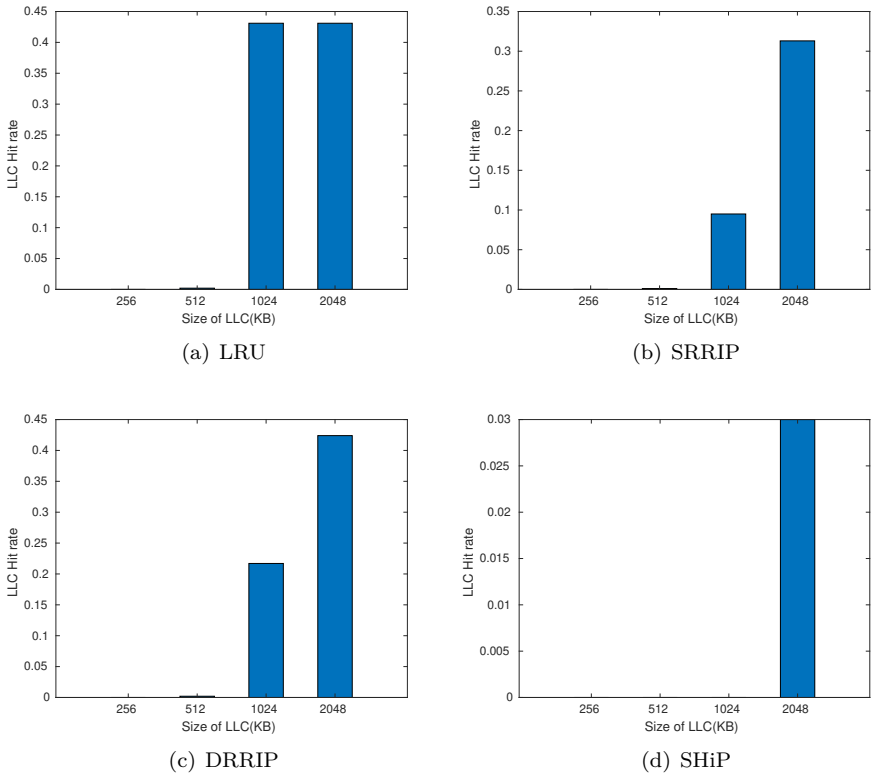
(d) SHiP Replacement policy

Figure 4.1: Varrying Associativity of cache

policy is giving better performance. For the cache size of 2048KB, the SRRIP policy has the same hit rate as the LRU policy. We can see that SRRIP is giving better performance than the DRRIP policy. This is maybe because of the thrashing access pattern of the given workload, where DRRIP dynamically uses the BRRIP policy over the SRRIP. The SHiP policy is giving very poor performance throughout the variations in the LLC size. This may be because it has failed to observe particular access patterns (called signatures) in the given synthetic workload.

From this part of the study, we observe that an increase in cache size and degree of associativity improves the hit rate of a cache. But we see that the improvement in hit rate is not the same in all the cases of replacement policies.

So we can say that cache hit rate not only depends on its parameters like cache size, associativity, block size, etc. But also depend upon the replacement policy that we are using and the different memory access patterns that the given workload follows.



(a) LRU (b) SRRIP

(c) DRRIP (d) SHiP

Figure 4.2: Varying LLC cache size

4.2 Performance trade-off in Asymmetric Multi core Architectures(AMPs)

In this section, we study the performance trade-off in Asymmetric multi-core architecture. Considering both performance and power as primary concerns,

we have implemented nine different configurations. In configurations 1 to 5, all cores are powerful cores with Out-of-order execution and an operating frequency of $2.66GHz$. But in the configurations 6 to 9, we have introduced Asymmetricity by changing the order of execution of cores(In-order or Out-of-order) and the frequency of operation of each core($1GHz$, $2.66GHz$). In these configurations, cores $0-7$ are In-order cores running at $1GHz$, whereas $8-15$ are Out-of-order cores running at $2.66GHz$. When Out-of-order and In-order cores share a cache memory, there is a chance that Out-of-order cores may occupy a large portion of the cache, which further degrades the performance of In-order cores running at a lower frequency. So we gradually partitioned the last level cache among the cores from fully shared to fully private. The remaining details of all the nine configurations are given in Table 4.3. The architecture used for our study, along with the corresponding simulation setup, is as given below.

Architecture used in this study

We have used Nehalem architecture, which is introduced by *Intel* in 2008. *Nehalem* is a microarchitecture that is both dynamically scalable and design-scalable. It regulates cache, threads, power, and cores dynamically to provide exceptional energy efficiency and performance on demand.

An overview of Nehalem Architecture

Nehalem has scalable performance for from 1 to 16(or more) threads and from 1 to 8(or more) cores. It has customizable system interconnects as well as a built-in memory controller. The three-level cache hierarchy of this microarchitecture which is shown in Fig 4.3 consists of 64KB of $L1$ cache with 32KB of data cache and 32KB of the instruction cache. Further, it has 256KB of $L2$ cache per core(private cache) for handling data and instructions. Finally, it has a fully inclusive and fully shared last level cache of size 8MB where all applications can use the entire cache. Nehalem has more out-of-order window and scheduler

size, which helps identify more independent operations that can run in parallel. Other buffers in the core have been increased in size to ensure that performance is not hampered.

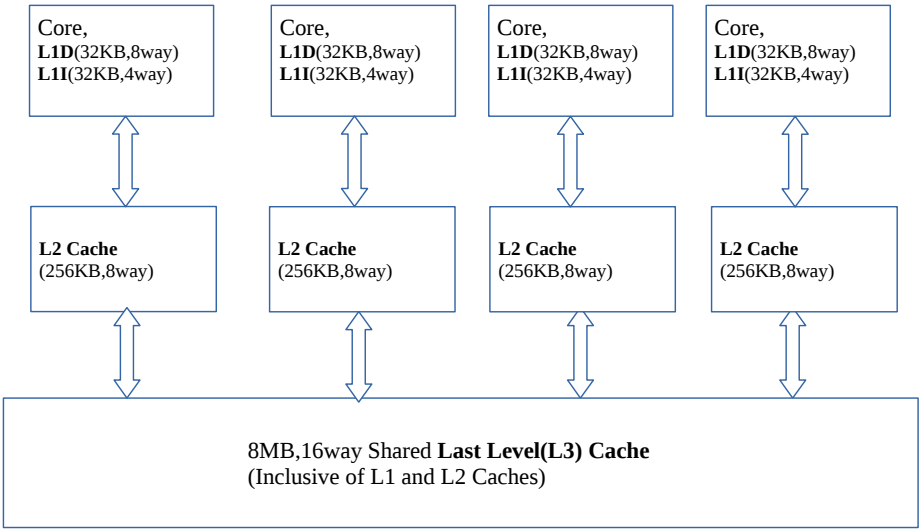


Figure 4.3: Cache hierarchy of Nehalem architecture with 4 cores

Simulator and Workloads used

We have used *Sniper* v7.3 simulator for our study.It is an $x86$ simulator that is fast, parallel, and accurate. It allows a range of flexible simulation options when exploring different homogeneous and heterogeneous multi-core architectures. Compared to other simulators, the textbf*Sniper* simulator allows one to run timing simulations for both multi-program workloads and multi-threaded, shared-memory programs with multiple cores at a very high speed.

In addition to these many features, it has the best base simulation infrastructure to simulate a more extensive set of workloads on more recent simulated hardware. We have used five different workloads in this study. They are Parsec-Bodytrack (Workload1), parsec-freqmine (Workload2), splash2-barnes (Work-

load3), parsec-fluidanimate (Workload4),splash2-radiosity(Workload5). Sniper integrates with the *McPAT*(Multicore Power, Area, and Timing) framework for power and area modeling of manycore architectures. We have used McPAT v1.0 in this study to get the different power consumption values of the processor.

Simulation Results:
In the initial stage, we have simulated all nine different configurations as mentioned in Table 4.3. We compared them with their L2 miss rate, L3 miss rate, and total power consumption which are shown in Table 4.4 & 4.5 and corresponding plots are shown in Fig.4.4. Except for $L2$ and $L3$ Cache levels, remaining all the parameters are common in all the configurations which are as mentioned in the Table. 4.2

Number of Cores	16
Benchmark	Parsec-Bodytrack
Inclusion Policy	Inclusive
Block Size	64 Bytes
Cache Coherence Protocol	mesi
Replacement Policy	LRU
$L1$ size	$32KB(I)$, $32KB(D)$
$L1$ shared cores	1(private to each core)
$L1$ Associtivity	$8(D)$, $4(I)$
$L2$ Size	$2048KB$(Total)
$L2$ Associtivity	8
$L3$ Size	$8192KB$(Total)
$L3$ Associtivity	16

Table 4.2: Configuration details

From Fig. 4.4(a), we observe that from configuration 1 to configuration 5, the $L2$ cache is changing from fully shared among all cores to fully private to each core. When we have more private caches, the chance of coherence misses is more, which increases the miss rate. Hence $L2$ Miss rate is increasing when moving from fully shared to fully private caches. But $L2$ cache is private to each core from configuration 5 to configuration 9, where the miss rate is almost the same.

Config Index	$L2$ **Cache Details**	$L3$ **Cache Details**	**Core Frequencies**
Config 1	All 16 cores sharing $2048KB$ of $L2$ cache	All 16 cores sharing $8192KB$ of $L3$ Cache	All 16 cores are running at $2.66GHz$
Config 2	Two sets of 8 cores and each set sharing $1024KB$ of $L2$ cache	all cores sharing $8192KB$ of $L3$ Cache	All cores are running at 2.66GHz
Config 3	Four sets of 4 cores, and each set sharing $512KB$ of $L2$ cache	all cores sharing $8192KB$ of $L3$ Cache	All cores are running at 2.66GHz
Config 4	eight sets 2 of cores, and each set sharing $256KB$ of $L2$ cache	all cores sharing $8192KB$ of $L3$ Cache	All cores are running at $2.66GHz$
Config 5	Each core with private $L2$ cache of $128KB$	all cores sharing $8192KB$ of $L3$ Cache	All cores are running at 2.66GHz
Config 6	Each core with private $L2$ Cache of size $128KB$	Two sets of 8 cores and each set sharing cache of size $4096KB$	Core $0-7$ are running at $1GHz$,Cores $8-15$ are running at $2.66GHz$
Config 7	Each core with private $L2$ Cache of size $128KB$	Four sets of 4 cores, and each set sharing Cache of size $2048KB$	Cores $0-7$ are running at $1GHz$,Cores $8-15$ are running at $2.66GHz$
Config 8	Each core with private $L2$ Cache of size $128KB$	eight sets of 2 cores, and each set sharing Cache of size $1024KB$	Cores $0-7$ are running at $1GHz$,Cores $8-15$ are running at $2.66GHz$
Config 9	Each core with private $L2$ Cache of size $128KB$	each core with private $L3$ Cache of size $512KB$	Cores $0-7$ are running at $1GHz$, Cores $8-15$ are running at $2.66GHz$

Table 4.3: L2 and L3 Cache details for different Configurations

Config Index	Total $L2$ Misses / Total $L2$ Accesses	$L2$ Miss rate
Config 1	1,775,452/32,658,514	5.43
Config 2	2,414,712/23,034,264	10.48
Config 3	3,464,822/24,173,825	14.33
Config 4	5,413,832/20,838,462	25.98
Config 5	17,851,930/20,985,015	85.06
Config 6	17,945,030/20,985,015	85.36
Config 7	18,105,468/21,045,636	85.99
Config 8	18,096,468/21,045,663	85.99
Config 9	18,013,786/20,914,704	86.13

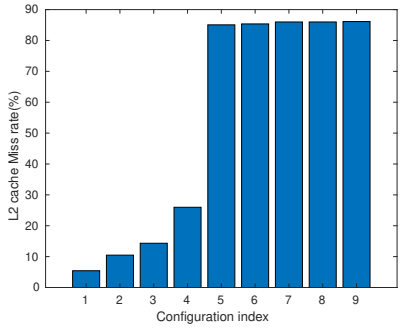
Table 4.4: $L2$ cache Miss rate for all configurations

Config Index	Total $L3$ misses/Total $L3$ Accesses	$L3$ Miss rate (%)	Total Power(W)
Config 1	83,872/1,792,716	4.67	275.238
Config 2	87,401/2,430,864	3.59	275.449
Config 3	87,561/3,481,529	2.51	274.293
Config 4	82,067/5,435,954	1.51	273.325
Config 5	85,948/17,871,357	0.48	273.718
Config 6	169,125/17,984,595	0.94	263.471
Config 7	309,358/18,176,373	1.7	265.63
Config 8	527,606/18,178,885	2.9	268.418
Config 9	1,033,091/18,161,730	5.688	270.978

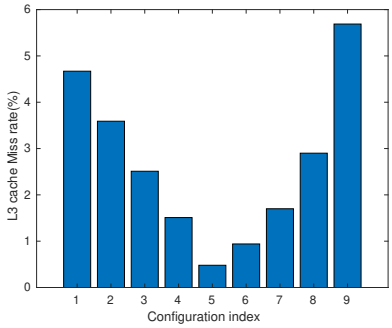
Table 4.5: $L3$ cache Miss rate and Power for all configurations

From Fig. 4.4(b), we can observe that when as we go from configuration 1 to 5, the L3 miss rate is decreasing even if the L3 cache is shared in all these cases. If we observe the number of total misses to the L3 cache in configurations 1 to 5 in the Table 4.5, they are almost the same. Due to the increasing miss rate of L2 shown in Table4.4, the number of accesses to the L3 cache increases, reducing the overall L3 miss rate. L3 cache is changing from a fully shared cache to a fully private cache in configurations 5 to 9, increasing the miss rate due to increased data inconsistency (coherent misses). When the number of private caches is increased, there is a chance for replicating the same data, which also increases the total miss rate due to inefficient utilization of net cache capacity.

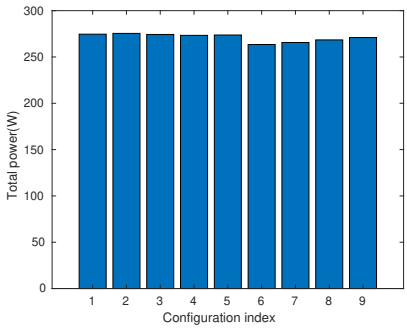
From Fig. 4.4(c), we observe that in Configurations 1 to 5, all cores are run-



(a) Miss Rate of $L2$ Vs Configuration Index



(b) Miss Rate of $L3$ Vs Configuration Index



(c) Total Power Vs Configuration Index

Figure 4.4: Performance trade-off in Heterogeneous Multi core Architecture

ning at 2.66GHz(Out-of-Order), so the total power consumed is almost the same in all the cases. But configurations 6 to 9 have eight cores running at 1GHz(In-order) and the other eight cores are running at 2.66GHz(Out-of-order), which gives low power consumption. So the power consumption is reduced when we go for asymmetric multi-cores. From the above observations, it is clear that when we go from all powerful cores(config 1) to asymmetric cores(config 9) miss rate of the L3 cache increased, whereas the total power consumption decreased.

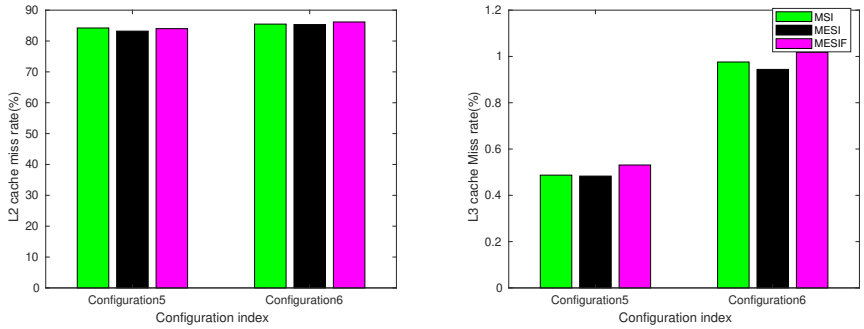
In Fig 4.4, we can see that both configurations 5,6 have the same L2 miss rate. But config 5 has a better L3 miss rate, whereas config 6 has better power consumption. Therefore, we wanted to simulate these two configurations with

the coherence protocols MSI, MESI, MESIF to check for any performance change as the miss rate here is primarily due to coherence.

4.2.1 Configuration $5, 6$ with different coherence protocols

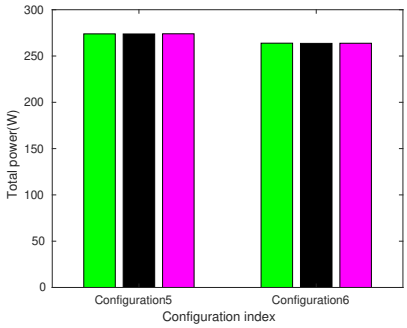
The configurations 5 and 6 are simulated against *MSI,MESI,MESIF* coherence protocols for L2 miss rate, L3 miss rate, and total power consumption of the processor. MSI is a write-invalidate coherence protocol. It has three possible states M(Modified), S(Shared), I(Invalid). Based upon each read and write operation, a cache block changes its state accordingly. The disadvantage of this protocol is when a processor changes a block that is not shared with any other processor, and if we want to read this data, we still need to search for that block in the cache of other processors. This increases the unnecessary bus traffic. So MESI protocol is introduced to deal with such kind of situation. This protocol uses four states where the three states M, S, I are the same as MSI protocol. An extra state E(Exclusive) is added in this case. But when a cache line, which multiple other processors hold in the S state, is requested, it will be served inefficiently by MESI protocol. It either initiates a memory request to fetch that particular line or ended up with redundant data due to the multiple responses by the caches having the same block. To address this issue, MESIF protocol is developed by intel. It contanis five different states. The first four states are the same as the MESI protocol, with an extra added state F(Forward). When we request for a line, it is only responded to by the cache with that line in F state and share it using direct cache to cache transfer.

From Fig. 4.5, we can observe that all the three coherence protocols are giving the same L2 miss rate and the same total power consumption. However, the L3 miss rate is also almost the same for all the protocols. But it is having better value against MESI than the other two protocols. So we can use the MESI protocol for better performance and power.



(a) L2 Cache Miss rate

(b) L3 Cache Miss rate



(c) Total Power

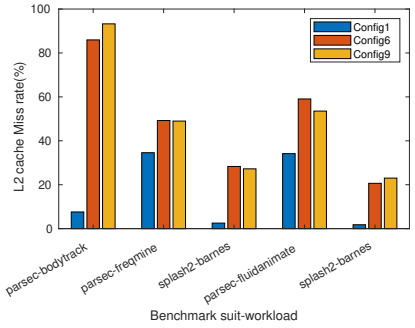
Figure 4.5: Configurations 5, 6 with different coherence protocols

4.2.2 Exploring the effect of shared and private caches

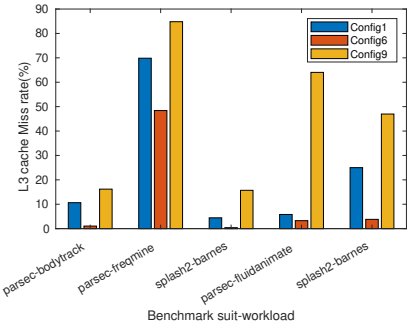
We have fixed the coherence protocol as MESI from the previous study for better performance and power. In section 4.1, we encountered a trade-off between L3 miss rate and total power consumption of Asymmetric multi-core architecture. Using configuration 1, configuration 6, and configuration 9, we want to investigate this trade-off further with three different replacement policies(LRU, MRU, Round Robin) using five different workloads(parsec-bodytrack, parsec-freqmine, splash2-barnes, parsec-fluidanimate, splash2-radiosity).

LRU replacement policy

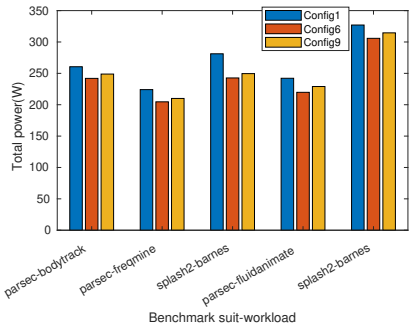
We have examined the L2 miss rate, L3 miss rate, and total power consumption in configurations $1, 6, 9$ against the replacement policy LRU with the five different workloads mentioned above. Corresponding results are as shown in Fig. 4.6



(a) L2 Cache Miss rate



(b) L3 Cache Miss rate

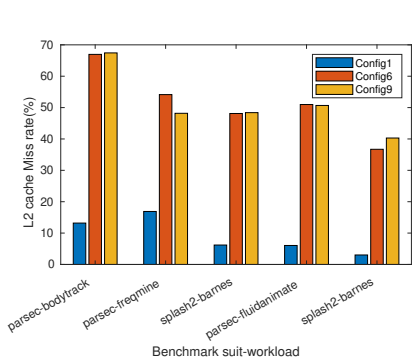


(c) Total Power

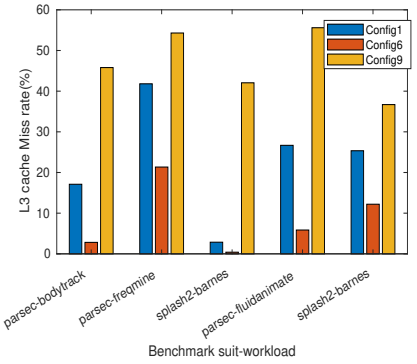
Figure 4.6: Configurations $1, 6, 9$ with LRU replacement policy

MRU replacement policy

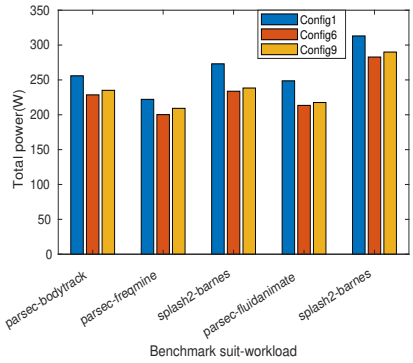
We have examined the L2 miss rate, L3 miss rate, and total power consumption in configurations $1, 6, 9$ against the replacement policy MRU with the five different workloads mentioned earlier. Corresponding results are shown in Fig. 4.7



(a) $L2$ Cache Miss rate



(b) $L3$ Cache Miss rate



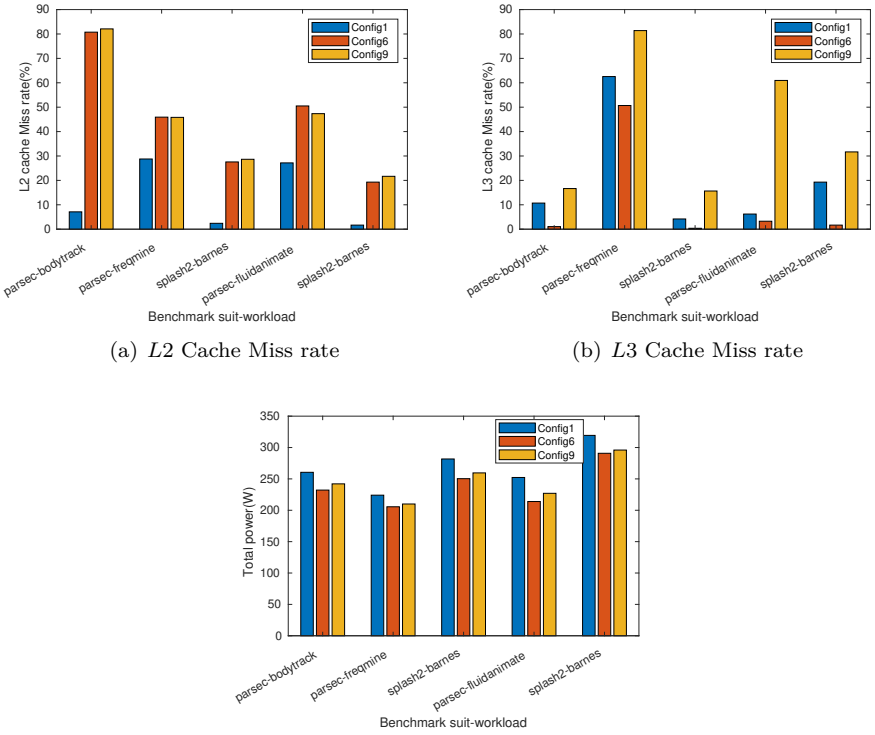
(c) Total Power

Figure 4.7: Configurations $1, 6, 9$ with MRU Replacement policy

Round Robin replacement policy

We have examined the L2 miss rate, L3 miss rate, and total power consumption in configurations $1, 6, 9$ against the replacement policy Round Robin with the

five different workloads mentioned above. Corresponding results are as shown in Fig. 4.8



(a) $L2$ Cache Miss rate

(b) $L3$ Cache Miss rate

(c) Total Power

Figure 4.8: Configuration $1, 6, 9$ with Round Robin Replacement policies

In the above figures, we can observe the trade-off between the L3 miss rate and total power consumption. Independent of the replacement policy and the workloads used when we go from configuration 1 to configuration 9, total power consumption decreases significantly in all the cases, but the L3 miss rate increases. Configuration 1 which has both L2,L3 shared caches is giving a better L2 cache miss rate than other configurations. But we need to consider the L3 misses over the L2 misses. Because as the misses to the L3 cache go to the main memory, which increases the overall latency and energy consumption. From configuration 1 to configuration 9, we observed the 13.25%, 20.25%, 20.4% increase

in L3 miss rate and 6.28%, 9.3%, and 7.7% reduction in total power consumption in LRU, MRU, and Round Robin replacement policies, respectively.

We have also observed that from configuration 1 to configuration 6 on average, there is 11.13% decrease in L3 miss rate and 10.54% reduction in power consumption. However, when we go from configuration 6 to configuration 9 there is 25% increase in L3 miss rate and 3.16% increase in power consumption. So, of all the configurations, configuration 6 provides the best performance and power.

Chapter 5

Conclusions and Future Work

This thesis has done a performance study of LLC(Last Level Cache) exploiting the heterogeneity for the Asymmetric multi-core architecture. We started our investigation by understanding the metrics that significantly affect the L2 and Last level caches by varying the parameters such as Cache size and associativity against different replacement strategies. We realized that replacement policy plays a significant impact on overall cache performance.

In the second part of our work, we understand Cache Hierarchy and simulate the AMP Architecture. It is crucial to mention beforehand that throughout our investigation of AMPs and their cache performance, the DVFS(Dynamic Voltage and Frequency Scaling) is kept enabled. We observe that the power variations in our results must not be considered on absolute terms; instead, they should be regarded as relative to each architecture. We begin our simulations with all cores operating on the same frequency, with each core sharing both the L2 and LL caches also have Out-of-order execution. Then we gradually partition the

caches such that they are private to each core and make some cores simpler, like lowering the frequency values and making them In-order to save on power. Our investigation showed the AMP Configuration 6 that has private L2 caches and different frequencies and order of execution is an optimal choice as it has the better power consumption compared to configuration 1 and also has the best LL miss rate compared to Configuration 9.We tested our architectures on multi-threaded Parsec and Splash2 benchmarks to have a real-world understanding.

AMPs have shown promise as the future architectures to solving several current computer architecture research problems. Investigations on proper task allocations to each core and even more diverse Cache Coherence Protocols could improve the performance of the existing systems by leaps and bounds. Also, the immediate next step to our work could be to increase the number of cores and bring more levels of heterogeneity in the system, such as flexible cache designs, varying the reorder buffer sizes, and propose many more optimizations.

Bibliography

[1] A. Vakil-Ghahani, S. Mahdizadeh-Shahri, M.-R. Lotfi-Namin, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Cache replacement policy based on expected hit count," *IEEE Computer Architecture Letters*, vol. 17, no. 1, p. 64–67, 2018.

[2] P.-Y. Peneau, D. Novo, F. Bruguier, G. Sassatelli, and A. Gamatie, "Performance and energy assessment of last-level cache replacement policies," *2017 First International Conference on Embedded Distributed Systems (EDiS)*, 2017.

[3] D. Ramtake, N. Singh, S. Kumar, and V. K. Patle, "Cache associativity analysis of multicore systems," *2020 International Conference on Computer Science, Engineering and Applications (ICCSEA)*, 2020.

[4] G. Jang and J.-L. Gaudiot, "Data shepherding: A last level cache design for large scale chips," *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019.

[5] K. Anandkumar, S. Akash, D. Ganesh, and M. S. Christy, "A hybrid cache replacement policy for heterogeneous multi-cores," *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2014.

[6] B. D. A. Silva, L. A. Cuminato, and V. Bonato, "Reducing the overall cache miss rate using different cache sizes for heterogeneous multi-core processors," *2012 International Conference on Reconfigurable Computing and FPGAs*, 2012.

[7] M. Sato, S. Nishimura, R. Egawa, H. Takizawa, and H. Kobayashi, "A cache partitioning mechanism to protect shared data for cmps," *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*, 2016.

[8] S. Gupta and H. Zhou, "Spatial locality-aware cache partitioning for effective cache sharing," *2015 44th International Conference on Parallel Processing*, 2015.

[9] F. Yuan and Z. Ji, "Replication-aware cache management for cmps with private llcs," *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, 2016.

[10] D. P. Kaur and V. Sulochana, "Design and implementation of cache coherence protocol for high-speed multiprocessor system," *2018 2nd IEEE International Conference on Power Electronics, Intelligent Control and Energy Systems (ICPEICES)*, 2018.

[11] A. Asaduzzaman and K. K. Chidella, "A novel directory based hybrid cache coherence protocol for shared memory multiprocessors," *2016 IEEE International Symposium on Phased Array Systems and Technology (PAST)*, 2016.

[12] S. Chakraborty and H. K. Kapoor, "Analysing the role of last level caches in controlling chip temperature," *IEEE Transactions on Sustainable Computing*, vol. 3, no. 4, p. 289–305, 2018.

[13] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache architecture for embedded systems," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, p. 136–146, 2003.

[14] A. D. Joshi, S. Indrajeet, N. Ramasubramanian, and B. S. Begum, "Analysis of multi-core cache coherence protocols from energy and performance perspective," *2017 International Conference on Recent Innovations in Signal processing and Embedded Systems (RISE)*, 2017.