A Project Report on

IMPLEMENTATION AND IMPROVISATION OF CACHE OPTIMIZATION TECHNIQUES

Submitted By

Ashwin S Sivan (B120223021)
Ayush Bhandari (B120223023)
Bharat Singh (B120223024)

A Project report submitted as a partial fulfillment towards the Term- II of

B.E. (Electronics & Telecommunication)

Savitribai Phule Pune University, Pune.

Guide
Prof. Shraddha Oza

Department of E&TC Engineering
Army Institute of Technology, Dighi, Pune-411 015

2017-18

C E R T I F I C A T E

This is to certify that

Ashwin S Sivan (B120223021)

Ayush Bhandari (B120223023)

Bharat Singh (B120223024)

of Army Institute of Technology, Dighi, Pune

have submitted project report on

*"**Implementation and Improvisation of***

***Cache Optimization Techniques**"*

as a partial fulfillment of Term –II for award of degree of Bachelor of E&TC, from Savitribai Phule Pune University, Pune, during the Academic Year 2017-18.

Project Guide	**Project Coordinator**	**HOD E&TC**
Prof. Shraddha Oza	Prof. Avinash Patil Prof. Renuka Bhandari	Prof. G R Patil

Acknowledgements

We are profoundly grateful to **Prof. Shraddha Oza** for her expert guidance and continuous encouragement throughout to see that this project rights its target since its commencement to its completion.

We would like to express deepest appreciation towards **Dr. B.P. Patil**, Principal, Army Institute of Technology, Pune, **Dr. G.R. Patil**, Head of Department of Electronics and Telecommunications Engineering and **Prof. Renuka Bhandari** and **Prof. Avinash Patil**, Project Coordinators whose invaluable guidance supported us in completing this project.

At last we must express our sincere heartfelt gratitude to all the staff members of Electronics and Telecommunications Department who helped us directly or indirectly during this course of work.

Ashwin S. Sivan
Ayush Bhandari
Bharat Singh

Abstract

Processor cache is an extremely important part of the modern computer and serves as a principle assistant in determining computers performance. The processor can make calculations only as quickly as it can be fed data. The job of the processor cache is to eliminate as many wasted cycles as possible and thereby achieving higher performance.

This project makes a thorough analysis of the cache optimization techniques through which higher performance in computers can be achieved.

Keywords: Cache, Performance and Optimization

Problem Statement

Today, one of the most prominent requirement of a programmer (or in general any user) is that his/her computer should be fast (in processing data and for general use). In general how fast a computer can be depends upon two factors - processor and RAM. Having a greater amount of RAM provides better multi-tasking.

Processor, on the other hand, is the part in a computer which is responsible for processing data, and the rate at which it can process data is known as clock rate. But processor can be as fast as it can be fed data, so that it can process that data faster. That's where Cache sweeps in. Cache is faster memory than hard drives and RAM, and is responsible for transferring data between processor, memory and different components of computer.

In this project we analyze different cache optimization techniques that can reduce cache hit time, reduce cache misses, has greater cache hits. We focus on improvising these optimization techniques so that these performance parameters can further be improved.

Contents

List of Figures

Chapter 1

Introduction

1.1 Principle of Locality

Principle of Locality states that program accesses a relatively smaller portion of their address space at any instant of time. Processors exploit this principle by using Temporal Locality (if an item is referenced, it tends to be referenced again soon, aka Locality in Time) and Spatial Locality (if an item is referenced items whose address are close by will tend to be referenced soon, aka Locality in Space), implementing them as a memory hierarchy, to achieve higher computing performance.

1.2 Memory Hierarchy

Memory Hierarchy is a structure of multiple levels of memory (with different speed and sizes), in which as the distance from processor increases, the access time and size of the memory both increases. This way the main goal, i.e. to present the user with as much memory as is available in the cheapest technology, while providing access at the speed offered by the fastest memory is achieved.

1.3 Cache

Cache, in its most general sense, is a name chosen to represent the level of memory hierarchy present between the processor and main memory. Cache is single handedly one of the most important part of computer, which determines its performance.

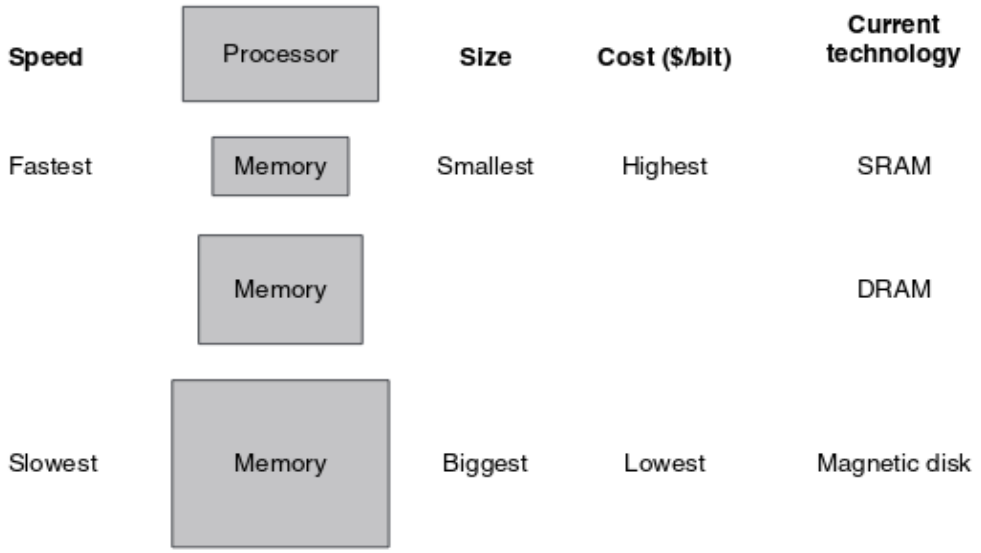

Figure 1.1: The basic structure of a memory hierarchy.

Chapter 2

Cache Optimization

2.1 Introduction

The memory components which are located between the processor core and main memory are called cache memories or **caches**. They are intended to contain copies of main memory blocks to speed up accesses to frequently needed data.The next lower level of the memory hierarchy is the main memory which is large but also comparatively slow. This chapter focuses on cache optimization techniques, for enhancing cache performance and its efficiency. This chapter also focuses on how these optimization techniques are implemented.

2.2 How it's done?

The average memory access formula gives the framework to present Cache Optimization for improving cache performance

$$Average\ Memory\ Access\ Time = Hit\ Time + Miss\ Rate \times Miss\ Penalty$$

Hence, we can categorize cache optimization techniques broadly into three categories :

1. Reducing Miss Rate
    - Larger Block Size
    - Large Cache Size
    - Higher Associativity
2. Reducing Hit Time

- Multilevel Caches
- Giving Reads priority over Writes

3. Reducing Miss Penalty

- Avoiding Address Translation during Cache Indexing

Chapter 3

Reducing Miss Rate

3.1 Introduction

Cache miss is a state where the data requested for processing by a component or application is not found in the cache memory. It causes execution delays by requiring the program or application to fetch the data from other cache levels or the main memory. If we can reduce the cache-miss rate, we can improve the performance of cache significantly.

3.2 What's a Block?

A minimum unit of information that may either be present or absent in cache.

3.3 Miss Categories

1. Compulsory - First time when we access the block
2. Capacity - If cache can't hold all blocks needed in a program
3. Conflict - If we use direct-mapped or set-associative strategy, two blocks may map to same record.

3.4 Compulsory Misses

These misses occur when a block is first referenced in an empty cache. Since, cache miss occurs, the block is immediately searched in lower level of memory (in memory hierarchy). As soon as the block is found, a copy of it is entered in the cache (for future references).

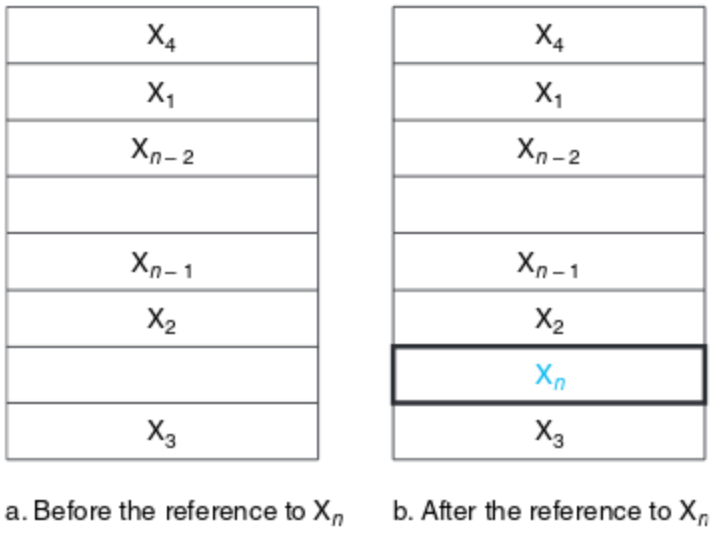

Figure 3.1: The cache just before and just after a reference to a word *Xn* that is not initially in the cache.

3.5 Capacity Misses

These misses occur when the blocks are being discarded from cache because cache cannot contain all blocks needed for program execution. The program working set (of blocks) is much larger than the capacity of cache.

3.6 Conflict Misses

In case of set associative or direct-mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set or block frame. These are also called as Collision Misses.

3.7 How to reduce these misses?

These misses are a function of cache parameters - Cache Size, Block Size and Associativity.

3.7.1 Reducing Compulsory Misses

As we increase the cache size, keeping the other two parameters constant, the number of potential compulsory misses will go up as there are more blocks to miss on in a cold cache.

The limit case is a cache with a single block - we will only get one compulsory miss. Increasing the block size means more adjacent words will be fetched on each miss, so references to these words will not cause compulsory misses - this exploits spatial locality.

Associativity only affects how cache blocks are arranged, not how they are fetched from main memory, so will not affect compulsory misses.

3.7.2 Reducing Capacity Misses

As we increase the cache size, more and more space would be available for all blocks that are needed for the program execution. Hence, no need to drop blocks and that ultimately results in reduced Capacity Misses.

Capacity misses are not really affected by block size because the decrease in the number of blocks that can be held is offset by their increased size.

Associativity has no effect on capacity misses as the total number of blocks remains the same no matter what the associativity.

3.7.3 Reducing Conflict Misses

Conflict misses are not affected by cache size since conflict misses arise from blocks from main memory mapping to the same position in the cache, which is mostly independent of the cache size.

Increasing the block size may increase the number of conflict misses. There is a greater chance to displace a useful block from the cache.

Type of associativity employed is significantly responsible for this type of miss. Employing higher degree of associativity solves the problem but results in higher Hit time.

	Cache Size	Block Size	Associativity
Compulsory Misses	0/-	+	0
Capacity Misses	+	0	0
Conflict Misses	0	0/-	+

The above table summarizes the effects that increasing the given cache parameters has on each type of miss. + means there's an improvement in the cache miss rate, 0 means no change and - means the situation gets worse.

3.8 Cache Size vs. Block Size

Larger blocks exploit spatial locality to lower miss rates. Increasing the block size usually decreases the miss rate. The miss rate may go up eventually if the block size becomes a significant fraction of the cache size, because the number of blocks that can be held in the cache will become small and there will be a great deal of competition for those blocks. As a result, a block will be bumped out of the cache before many of its word are accessed. Stated alternatively, spatial locality among the words in a block decreases with a very large block; consequently the benefits in the miss rate becomes smaller.

A more serious problem associated with just increasing the block size is that the cost of the miss increases. The miss penalty is determined by the time required to fetch the block from the next lower level of the hierarchy and load it into the cache. The time to fetch the block the block has two parts: the latency to the first word and transfer time for the rest of the rest of the block. Clearly, unless we change the memory system, the transfer time - and hence the miss penalty - will likely to increase as the block size increases. Furthermore, the improvement in miss rate starts to decrease as the blocks becomes larger. The result is that the increase in the miss penalty overwhelms the decrease in the miss rate for blocks that are too large, and the cache performance thus decreases.

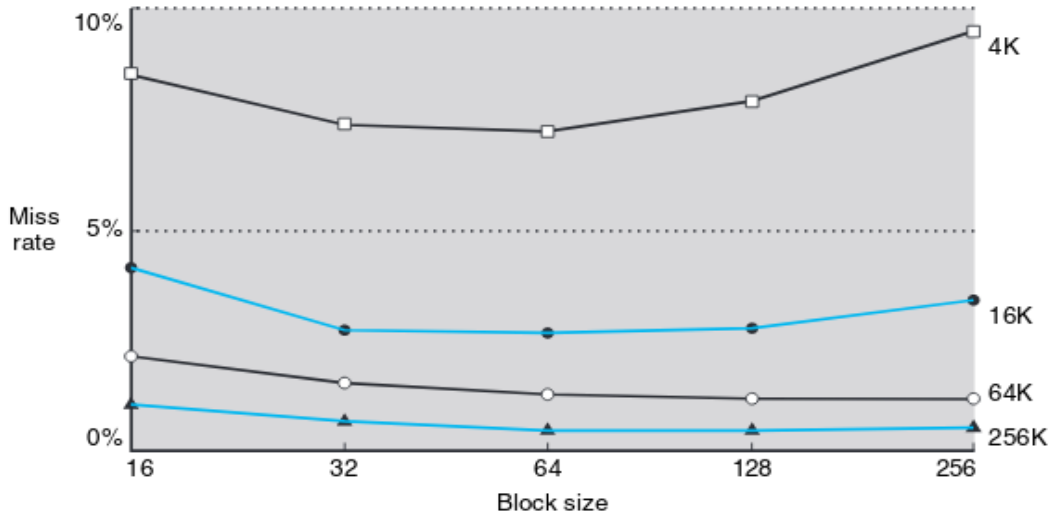

Figure 3.2: Miss rate versus block size.

3.9 Reducing Cache Misses by More Flexible Placement of Blocks

There is actually a whole range of schemes available for placing blocks. Choosing a block placement technique according to a particular application can significantly make changes in Cache Optimization.

3.9.1 Direct Mapped Cache

In a direct-mapped cache, the position of a memory block is given by

$$Index = Block\ number\ \%\ Number\ of\ blocks\ in\ the\ cache$$

A block can go in exactly one place in the cache. There is a direct mapping from any block address in memory to a single location in upper level of the hierarchy.

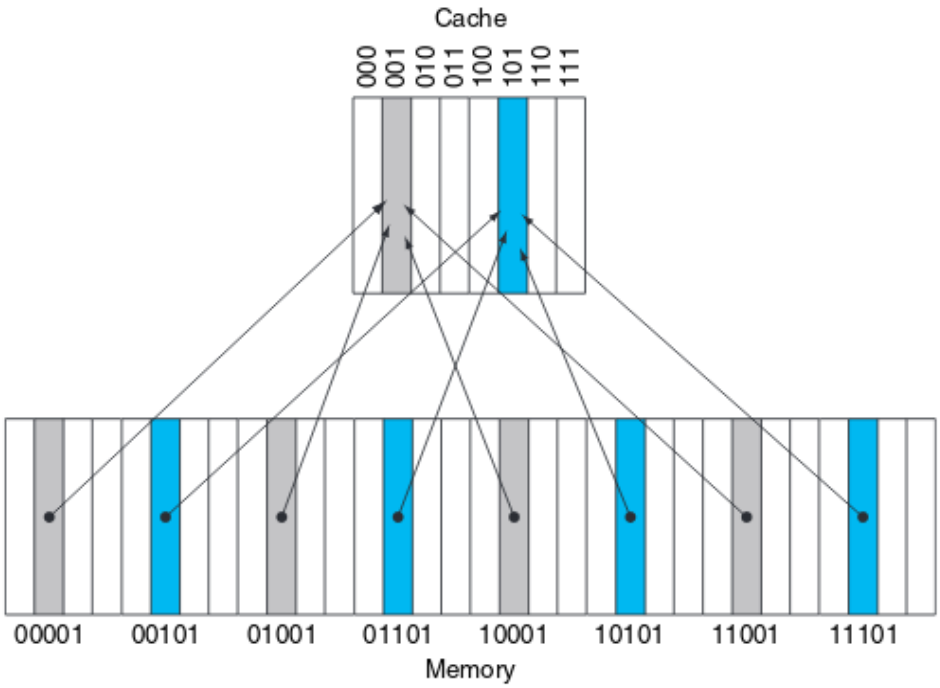

Figure 3.3: A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations.

3.9.2 Fully Associative Cache

In this scheme, a block can be placed in any location in the cache. To find a given block in a fully associative cache, all entries in the cache must be searched because a block can be placed in any one. To make the search practical, it's done in parallel with a comparator associated with each cache entry. These comparators significantly increase the hardware cost, effectively making fully associative placement practical only with small number of blocks.

3.9.3 Set-Associative Cache

In this scheme, there are a fixed number of locations where each block can be placed. A set-associative cache with n locations for a block is called an n-way set-associative cache consists of a number of sets, each of which consists of n blocks. Each block in the memory maps to unique set in the cache given by the index field, and a block can be placed in any element of that set. Thus it combines both direct-mapped and fully associative schemes: a block is directly mapped into a set, and then all the blocks in the set are searched for a match.

The main advantage of increasing the degree of associativity is that it usually

decreases the miss rate. The main disadvantage is a potential decrease in hit time.

Figure 3.4: The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement.

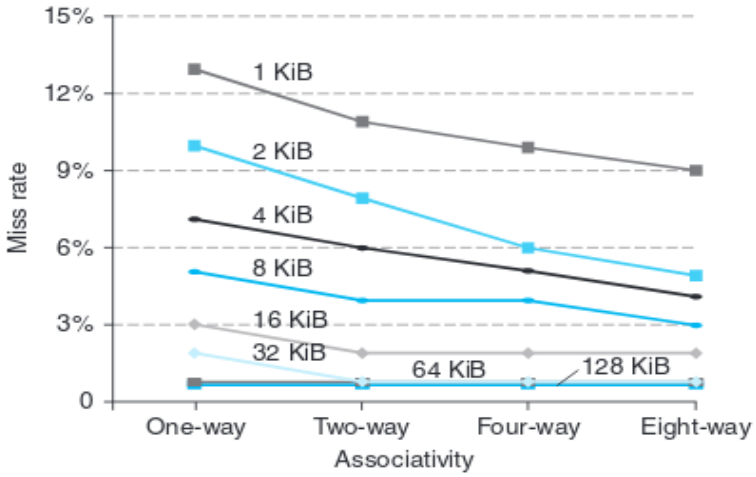

Figure 3.5: The data cache miss rates for each of eight cache sizes improve as the associativity increases.

Chapter 4

Reducing Hit Time

4.1 Introduction

The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss is known as **Hit Time.** In this chapter we discuss techniques by which we reduce this hit time, and hence improve the performance of cache significantly.

4.2 Multilevel Caches

All modern computers make use of caches. To close the gap further between the fast clock rates of modern processors and the increasingly long time required to access DRAMs, most microprocessors support an additional level of caching. This second-level cache is normally on the same chip and is accessed whenever a miss occurs in the primary cache. If the second-level cache contains the desired data, the miss penalty for the first-level cache will be essentially the access time of the second-level cache, which will be much less than the access time of main memory. If neither the primary nor the secondary cache contains the data, a main memory access is required, and a larger miss penalty is incurred.

The design considerations for a primary and secondary cache are significantly different, because the presence of the other cache changes the best choice versus a single-level cache. In particular, a two-level cache structure allows the primary cache to focus on minimizing hit time to yield a shorter clock cycle or fewer pipeline stages, while allowing the secondary cache to focus on miss rate to reduce the penalty of long memory access times.

The effect of these changes on the two caches can be seen by comparing each

cache to the optimal design for a single level of cache. In comparison to a single-level cache, the primary cache of a multilevel cache is often smaller. Furthermore, the primary cache may use a smaller block size, to go with the smaller cache size and also to reduce the miss penalty. In comparison, the secondary cache will be much larger than in a single-level cache, since the access time of the secondary cache is less critical. With a larger total size, the secondary cache may use a larger block size than appropriate with a single-level cache. It often uses higher associativity than the primary cache given the focus of reducing miss rates.

Disadvantage: Multilevel caches create several complications. First, there are now several different types of misses and corresponding miss rates. There is no general way to calculate overlapped miss latency, so evaluations of memory hierarchies for out-of-order processors is very complex.

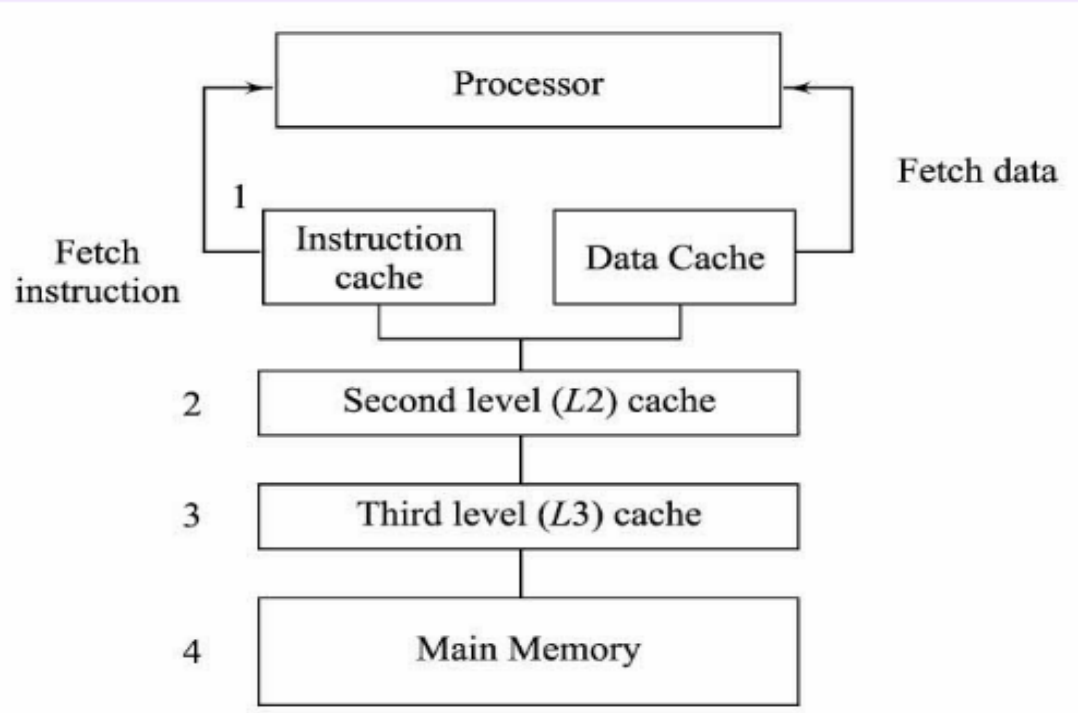

Figure 4.1: Multilevel Cache

4.3 Handling Writes

Writes work differently than read. Suppose on a store instruction, we wrote the data into only the data cache (without changing main memory); then, after the write into the cache, memory would have a different value from that in the cache. In such a case, the cache and memory are said to be *inconsistent*.

There are two most popular schemes available for writes - ***Write-Through*** and

Write-Back. They are explained in detail below.

4.3.1 Write-Through

A scheme in which writes always update both the cache and the next lower level of the memory hierarchy, ensuring that data is always consistent between the two.The other key aspect of writes is what occurs on a write miss. We first fetch the words of the block from memory. After the block is fetched and placed into the cache, we can overwrite the word that caused the miss into the cache block. We also write the word to main memory using the full address.

Disadvantage: Although this design handles writes very simply, it would not provide very good performance. With a write-through scheme, every write causes the data to be written to main memory. These writes will take a long time, likely at least 100 processor clock cycles, and could slow down the processor considerably. For example, suppose 10% of the instructions are stores. If the CPI without cache misses was 1.0, spending 100 extra cycles on every write would lead to a CPI of $1.0 + 100 \times 10\% = 11$, reducing performance by more than a factor of 10.

Write Buffer: It's a queue that holds data while the data is waiting to be written to memory. A write buffer stores the data while it is waiting to be written to memory. After writing the data into the cache and into the write buffer, the processor can continue execution. When a write to main memory completes, the entry in the write buffer is freed. If the write buffer is full when the processor reaches a write, the processor must stall until there is an empty position in the write buffer.

4.3.2 Write-Back

In a write-back scheme, when a write occurs, the new value is written only to the block in the cache. The modified block is written to the lower level of the hierarchy when it is replaced. Write-back schemes can improve performance, especially when processors can generate writes as fast or faster than the writes can be handled by main memory; a write-back scheme is, however, more complex to implement than write-through.

4.3.3 MESI Protocol

MESI Protocol is a cache synchronization or cache-coherence protocol and it is one

of the most popular and common protocol that supports write-back scheme. By using write back caches, we save a lot on bandwidth which is generally wasted on a write through cache. There is always a dirty state present in write back caches which indicates that the data in the cache is different from that in main memory. This Protocol requires cache to cache transfer on a miss if the block resides in another cache. This protocol reduces the number of main memory transactions and hence makes a significant improvement in cache performance.

1. **Modified:** The cache line is present only in the current cache, and is dirty - it has been modified(M state) from the value in main memory.
2. **Exclusive:** The cache line is present only in the current cache, but is clean - it matches main memory.
3. **Shared:** Indicates that this cache line may be stored in other caches of the machine and is clean - it matches the main memory.
4. **Invalid:** Indicates that this cache line is invalid (unused).

4.4 Giving Reads Priority Over Writes

This optimization serves reads before writes have been completed. We start with looking at complexities of a write buffer. With a write-through cache the most important improvement is a write buffer of the proper size. Write buffers, however, do complicate memory accesses in that they might hold the updated value of a location needed on a read miss.

The simplest way out of this is for the read miss to wait until the write buffer is empty. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue. Virtually all desktop and server processors use the latter approach, giving reads priority over writes. The cost of writes by the processor in a write-back cache can also be reduced. Suppose a read miss will replace a dirty memory block. Instead of writing the dirty block to memory, and then reading memory, we could copy the dirty block to a buffer, then read memory, and then write memory. This way the CPU read, for which the processor is probably waiting, will finish sooner. Similar to the situation above, if a read miss occurs, the processor can either stall until the buffer is empty or check the addresses of the words in the buffer for conflicts.

Chapter 5

Reducing Miss Penalty

5.1 Introduction

Time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, insert it in the level that experienced the miss, and then pass the block to the requestor, is known as **Miss Penalty**. In this chapter we discuss techniques by which we can reduce this Miss Penalty and hence improve the performance of cache significantly.

5.2 Virtual Memory

It's a technique in which main memory can act as a cache for secondary storage, usually implemented with magnetic disks.

Motivation: Historically there were two major motivation for virtual memory - to allow efficient and safe sharing of memory among multiple programs, such as for the memory needed by multiple virtual machines for cloud computing, and to remove the programming burdens of a small, limited amount of main memory.

Virtual memory implements the translation of a programs address space to physical addresses . This translation process enforces protection of a programs address space from other virtual machines. A virtual memory block is called a page , and a virtual memory miss is called a page fault .

With virtual memory, the processor produces a **virtual address** , which is translated by a combination of hardware and software to a **physical address** , which in

turn can be used to access main memory. This process is called as **Address Translation**.

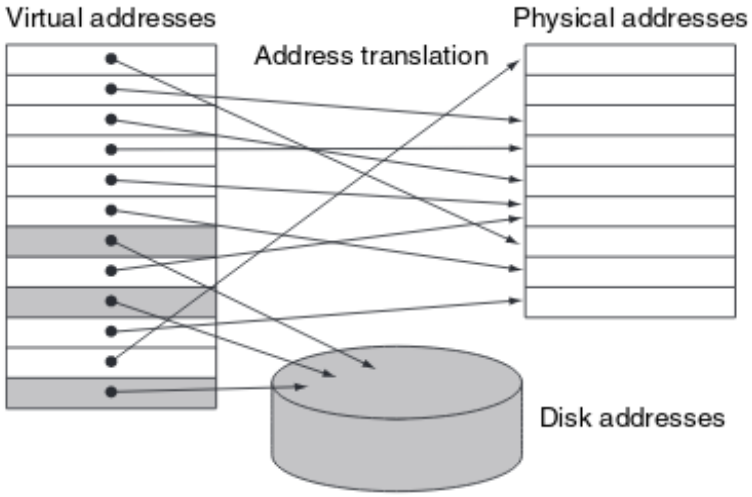

Figure 5.1: In virtual memory, blocks of memory (called pages) are mapped from one set of addresses (called virtual addresses) to another set (called physical addresses).

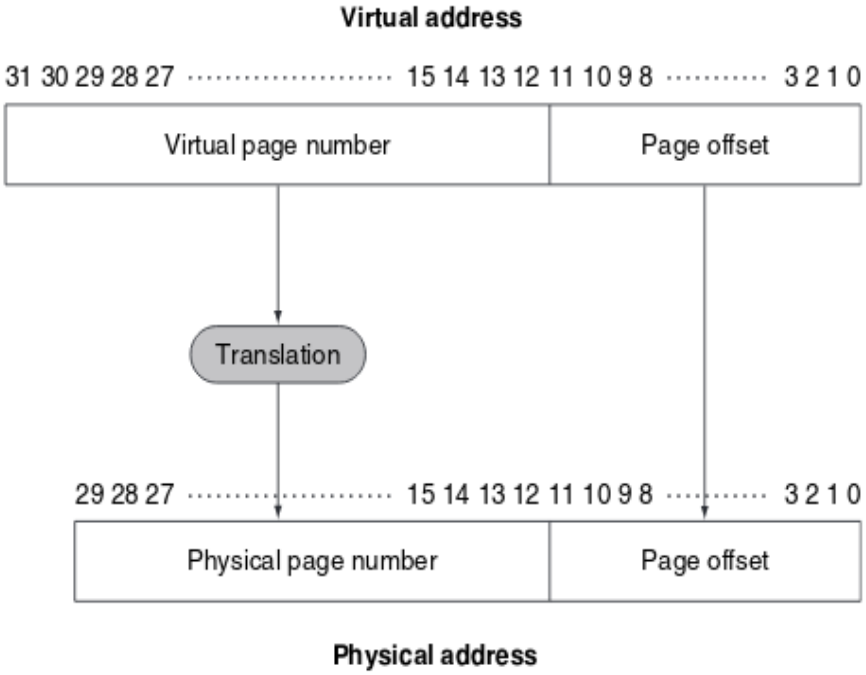

Figure 5.2: Mapping from a virtual to a physical address.

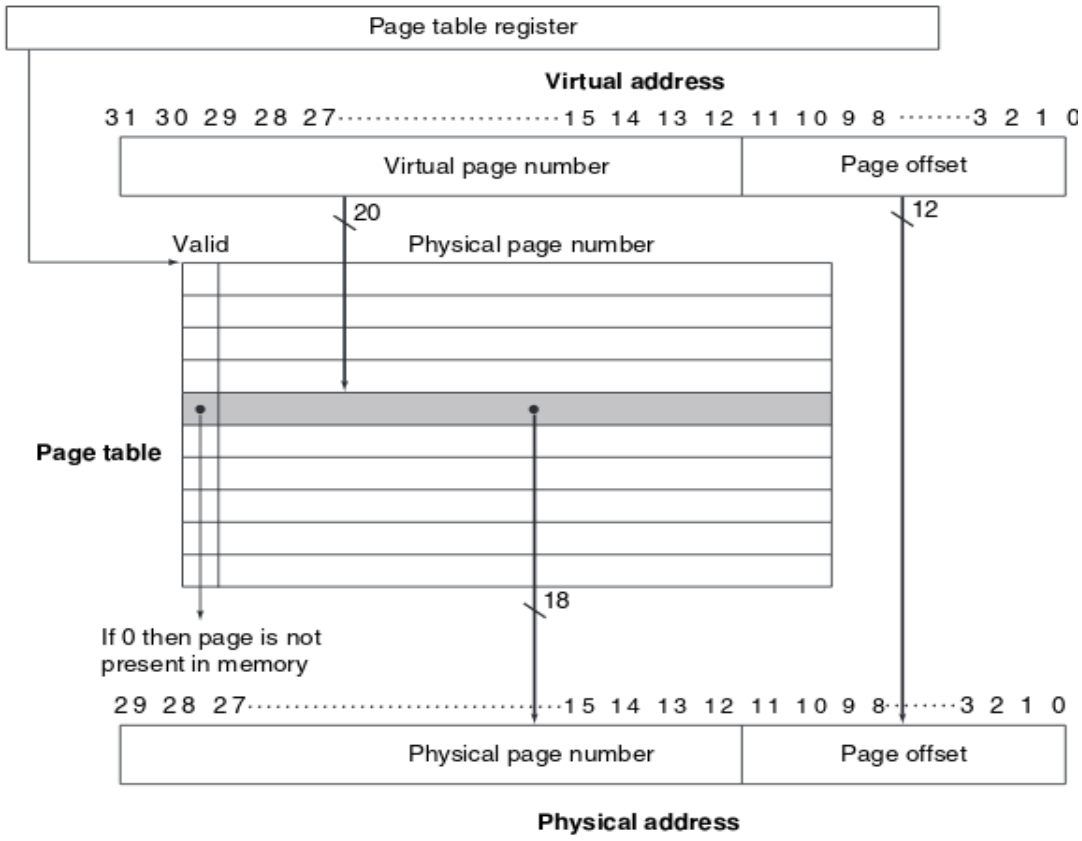

Figure 5.3: The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address.

5.3 Reducing Miss Penalty using TLB

Since the page tables are stored in main memory, every memory access by a program can take at least twice as long: one memory access to obtain the physical address and a second access to get the data. The key to improving access performance is to rely on locality of reference to the page table. When a translation for a virtual page number is used, it will probably be needed again in the near future, because the references to the words on that page have both temporal and spatial locality.

5.3.1 Translation-Lookaside Buffer (TLB)

Modern processors include a special cache that keeps track of recently used translations. This special address translation cache is traditionally referred to as a translation-lookaside buffer (TLB), although it'd be more accurate to call it a Translation Cache. Each tag entry in the TLB holds a portion of the virtual page number, and each data entry of the TLB holds a physical page number.

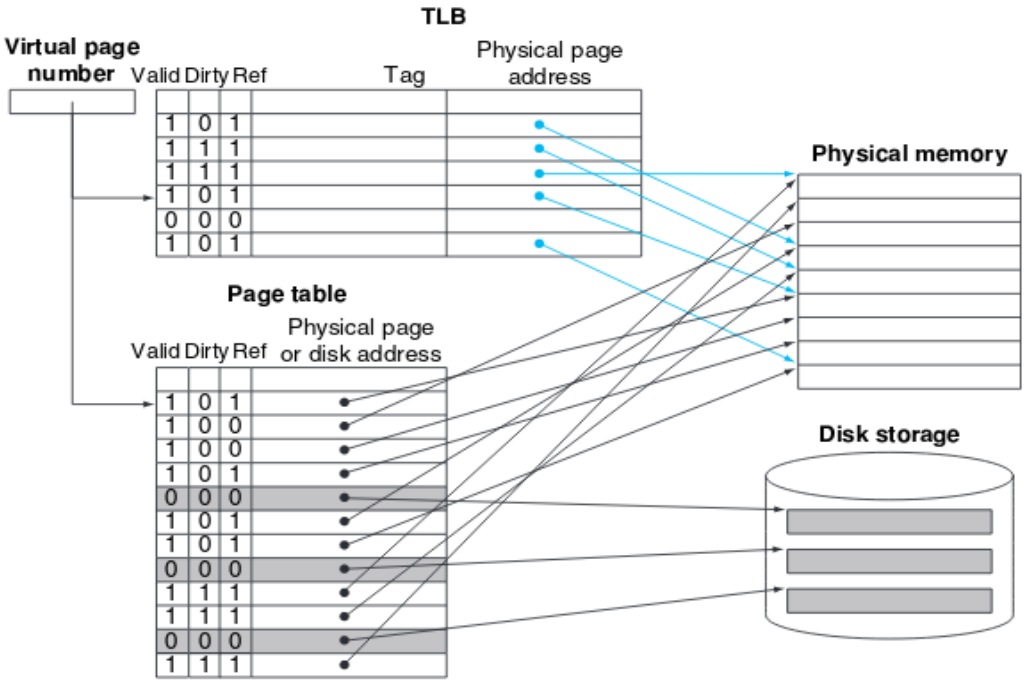

Figure 5.4: The TLB acts as a cache of the page table for the entries that map to physical pages only.

On every reference, we look up the virtual page number in the TLB. If we get a hit, the physical page number is used to form the address, and the corresponding reference bit is turned on. If the processor is performing a write, the dirty bit is also turned on. If a miss in the TLB occurs, we must determine whether it is a page fault or merely a TLB miss. If the page exis ts in memory, then the TLB miss indicates only that the translation is missing. In such cases, the processor can handle the TLB miss by loading the translation from the page table into the TLB and then trying the reference again. If the page is not present in memory, then the TLB miss indicates a true page fault. In this case, the proces sor invokes the operating system using an exception. Because the TLB has many fewer entries than the number of pages in main memory, TLB misses will be much more frequent than true page faults.

After a TLB miss occurs and the missing translation has been retrieved from the page table, we will need to select a TLB entry to replace. Because the reference and dirty bits are contained in the TLB entry, we need to copy these bits back to the page table entry when we replace an entry. These bits are the only portion of the TLB entry that can be changed. Some systems use other techniques to approximate the reference and dirty bits, eliminating the need to write into the TLB except to load a new table entry on a miss.

Chapter 6

Victim Cache And Adaptive Cache

6.1 Introduction

Projections of computer technology forecast processors with very high peak performances in relative near future. These processors could easily lose half or more of their performance in the memory hierar- chy if the hierarchy design is based on conventional caching techniques. Victim caching is an improvement to miss caching that loads the small fully-associative cache with the vic- tim of a miss and not the requested line. Small victim caches of 1 to 5 entries are even more effective at removing conflict misses than miss caching.

6.2 Victim Caches

Victim cache is a small full associative cache placed in the refill path of CPU to improve the performance. Miss caching places a fully-associative cache between cache and its re-fill path. Misses in the cache that hit in the miss cache have a one cycle penalty, as opposed to a many cycle miss penalty without the miss cache. Victim Caching is an improvement to miss caching that loads the small fully-associative cache with victim of a miss and not the requested cache line.

A victim cache is a hardware cache designed to decrease conflict misses and improve hit latency for direct-mapped caches. It is employed at the refill path of a Level 1 cache, such that any cache line which gets evicted from the cache is cached in the victim cache. Thus, the victim cache gets populated only when data is thrown out of Level 1 cache. In case of a miss in Level 1, victim cache is looked up. If the resulting access is a hit, the contents of the Level 1 cache-line and the matching victim cache line are swapped.

Consider a system with a direct-mapped cache and a miss cache. When a miss occurs, data is loaded into both the miss cache and the direct-mapped cache. In a

sense, this duplication of data wastes storage space in the miss cache. The number of duplicate items in the miss cache can range from one (in the case where all items in the miss cache map to the same line in the direct-mapped cache) to all of the entries (in the case where a series of misses occur which do not hit in the miss cache).

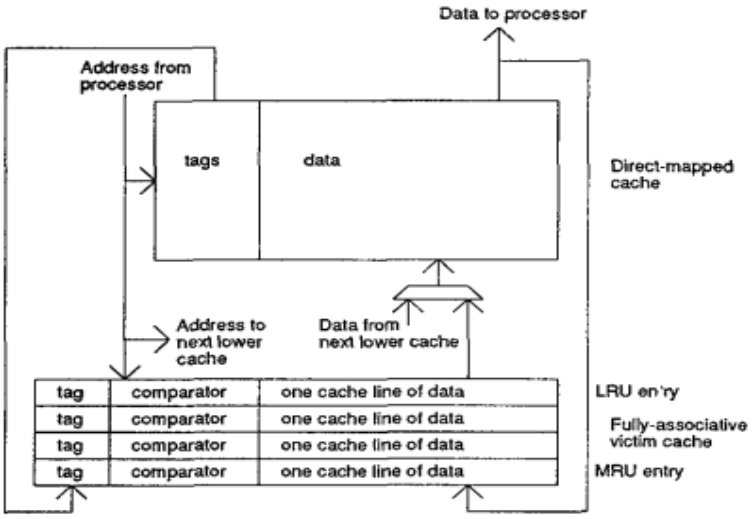

Figure 6.1: Victim Cache Organization

The percentage of conflict misses removed by victim caching is given in Figure below. Note that victim caches consisting of just one line are useful, in contrast to miss caches which must have two lines to be useful. All of the benchmarks have improved performance in comparison to miss caches, but instruction cache performance and the data cache performance of benchmarks that have conflicting long sequential reference streams (e.g., ccom and linpack) improve the most.

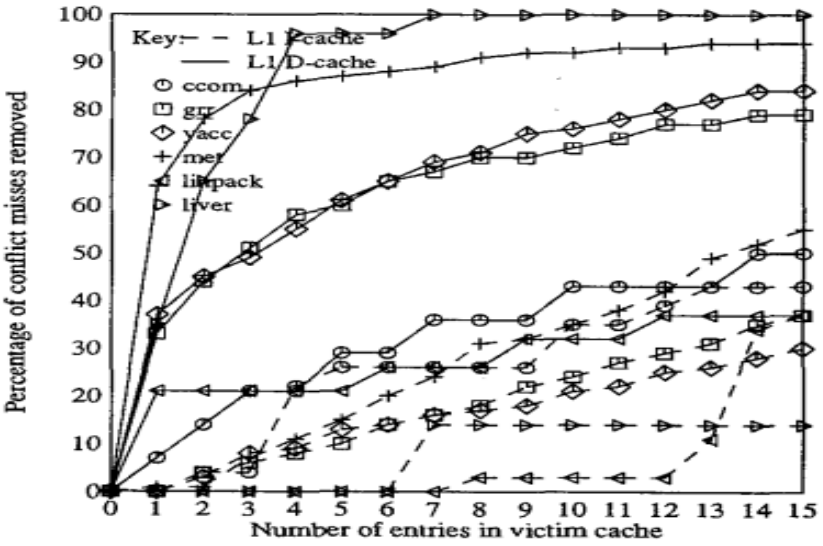

Figure 6.2: Conflict misses removed by victim caching

6.3 Adaptive Caches

Adaptive Cache evaluates the idea of adaptive processor cache management. Specifically, general scheme by which we can combine any two cache management algorithms (e.g., LRU, LFU, FIFO, Random) and adaptively switch between them, closely tracking the locality characteristics of a given program. Similar works in virtual cache management at Operating Systems level suggests that it is possible to adapt over two replacement policies to provide an aggregate policy that always performs within a constant factor of the better component policy.

The goal of adaptive caching is to switch on-the-fly between two policies, in order to imitate the one that has been performing better on recent memory accesses. To do this, the adaptive cache needs to maintain some more information than a traditional cache. Specifically, our adaptive cache design has two extra elements (shown in Figure below) compared to traditional caches.

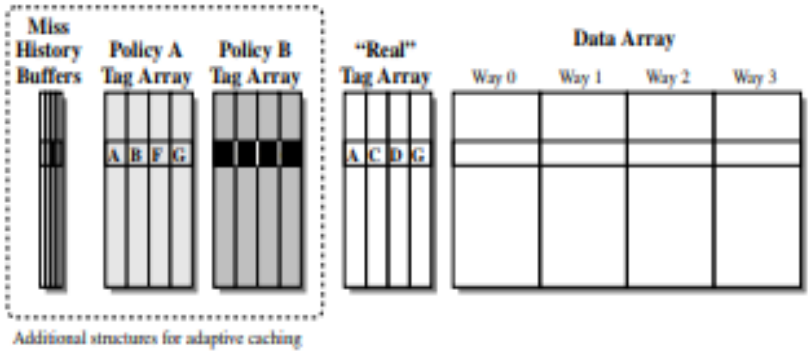

Figure 6.3: The basic hardware organization of an adaptive cache. The additional components required for the technique are grouped on the left.

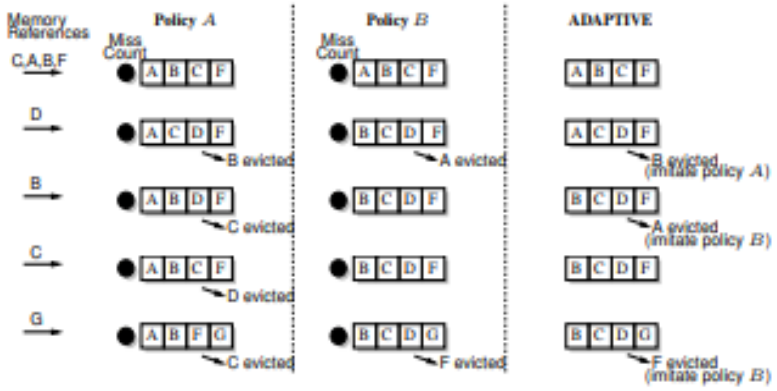

Figure 6.4: Example adaptive cache behavior

6.4 Adaptive Cache Replacement Algorithm

```
if ( misses(A) > misses(B) ) // if A is missed more than B
{
    // then imitate B
    if ( B missed AND the block it evicts
        is in adapt cache )
        adaptive cache evicts the same block B evicts
    else
        adaptive cache evicts any block not in B
    // such a block is guaranteed to exist,since,in this
    // case,Bs cache contents are different from those
    // in the adaptive cache and the two caches have
    // the same size.
}
else // B missed more than A
    Same as above, but with B replaced by A
```

Chapter 7

Proposed Cache Architecture Design

7.1 Introduction

Victim cache and Adaptive Cache provide improvement in Cache performance. Individually they have their advantages like reduced hit time, high hit rate, low miss rate etc. In our proposed cache design we combine both these approaches to harness best of both worlds in our cache design.

Advantages of using Victim Cache:

1. Lower search time (VC is smaller in size)
2. Provides associativity to L1 cache (VC is fully-associative)
3. It contains blocks evicted recently from L1, hence avoids duplicacy

Advantages of using Adaptive Cache Replacement:

1. High Hit Ratio
2. Uses best replacement techniques
3. Provides at-least worst case guarantee

7.2 Proposed Cache Design

We combine these two approaches to the Cache architecture, which enables us to gain performance enhancements and advantages mentioned above.

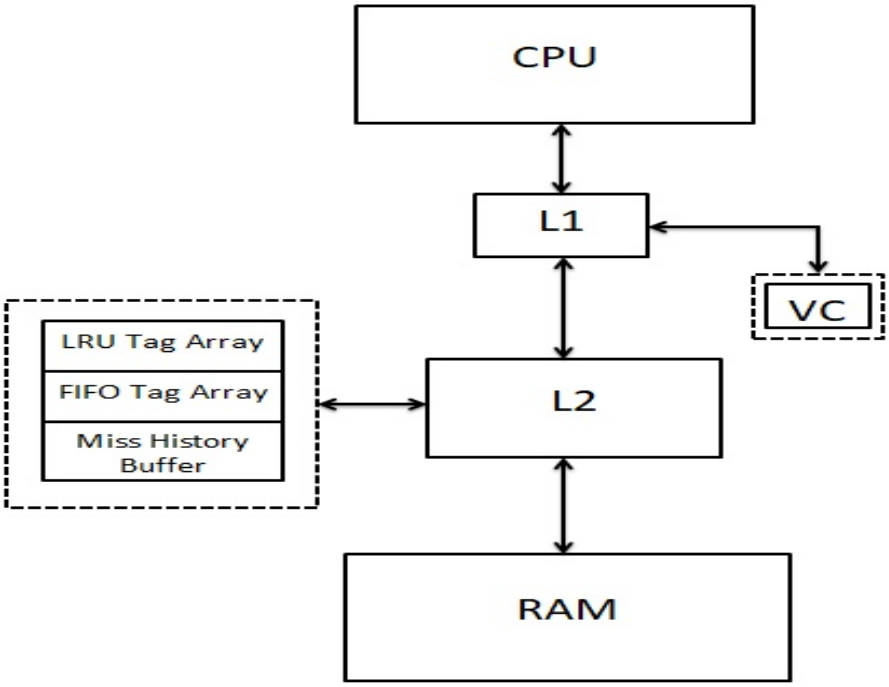

Figure 7.1: Structure of our proposed Cache Architecture

Chapter 8

System-Flow Diagram

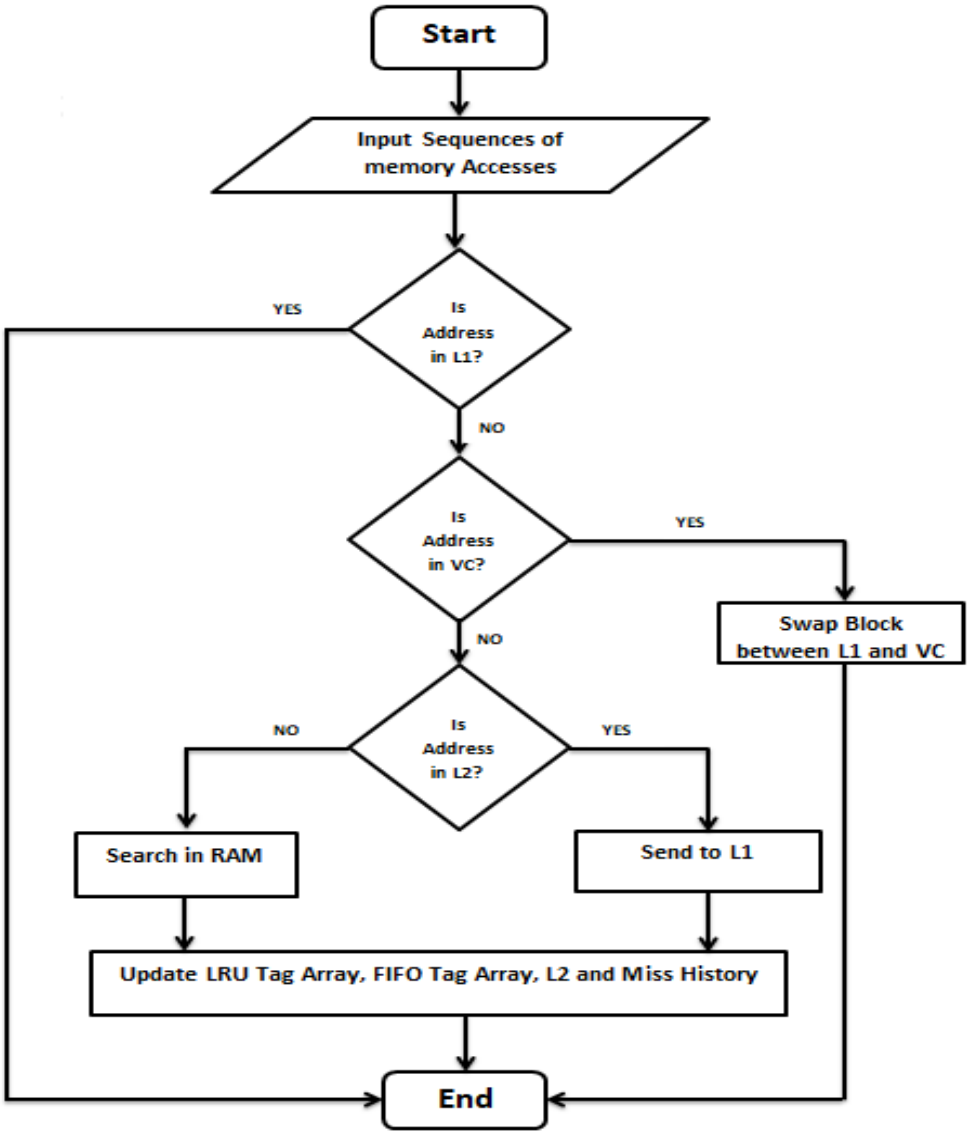

Figure 8.1: System-Flow Diagram

Chapter 9

Methodology

9.1 Software and Programming Language

- Spyder IDE
- Python
- C
- Vim Editor
- GCC

9.2 Cache Simulations

9.2.1 Introduction

These cache simulations simulate the behaviour of a normal cache, cache with victim cache present in the hierarchy and cache in which victim cache is present in hierarchy and adaptive cache replacement applied to L2 cache. File `Random3.txt` contains a sample of accesses requested by CPU (at random).

Another simulation simulates the behaviour of a Direct Mapped Cache under different write techniques (i.e. Write Back and Write Through). This simulation is written in C programming language. Files `trace0.txt`, `trace1.txt`, `trace2.txt` and `trace3.txt` are trace files that contain memory access requested by CPU. Each line is for one memory access. First column of is the address of the instruction that is causing memory access, R (W) indicates whether the operation is Read (Write) and the last column is the memory address that is being accessed.

9.2.2 Algorithm Used For Simulating Our Proposed Cached Model

1. CPU requests for a block in memory (from our trace file)
2. Check if the block is present in L1 Cache
3. If present copy the block from L1 Cache to CPU Registers
4. If the block is not present in L1 Cache, search for block in Victim Cache
5. If present in Victim Cache and swap with the element evicted from L1 Cache
6. If requested block isn't present in Victim Cache, look for block in Cache L2
7. If present in L2 Cache, copy block in L1 Cache
8. If requested block isn't present in L2 Cache look for block in RAM
9. If present copy block in L2 according to Adaptive Cache Replacement Algorithm and L1 Cache
10. If requested block isn't present in RAM, look for block in Secondary Memory
11. If present, copy block in RAM, L2 Cache (using ACR) and L1 cache
12. If requested block isn't present in Secondary Storage, it's a bad request

9.2.3 Algorithm Used For Simulating Behaviour of Direct-Mapped Cache Under Write-Back And Write-Through Scheme

General Algorithm

1. Validate inputs.
2. Open the trace file for reading.
3. Create a new cache object.
4. Read a line from the file.
5. Parse the line and read or write accordingly.
6. If the line is #eof continue, otherwise go back to. step 4
7. Print the results.
8. Destroy the cache object.
9. Close the file.

Write Algorithm

1. Validate the inputs.
2. Convert the hexadecimal address to a parsable binary format.
3. From the formatted binary string, extract the tag and index.
4. Convert the index to an integer corresponding to a slot in the array.
5. If the current block is valid and has the same tag as the one we are searching for, increment the cache hits, and mark the block as dirty. In addition, if the write policy is write through, increment the writes. Go to step 8.
6. Increment the misses and the reads.
7. If the write policy is write back and the block is dirty, increment the writes.
8. Set the block to valid, dirty, and store the tag in it.

Read Algorithm

1. Validate the inputs.
2. Convert the hexadecimal address to a parsable binary format.
3. From the formatted binary string, extract the tag and index.
4. Convert the index to an integer corresponding to a slot in the array.
5. If the current block is valid and has the same tag as the one we are searching for, increment the cache hits, and then go to step 8.
6. Otherwise, increment the misses and reads values.
7. If the write policy is write back and the block is marked as dirty, increment the writes and mark the block as clean.
8. Set the block as valid and store the tag within the block.

Chapter 10

Experimentation

10.1 Simulation of Direct-Mapped Cache

The program can be invoked as

```
sim < write-policy > < trace file >

< write policy > is one of
        wt (write through)
        wb (write back)

< trace file > name of trace file having memory accesses.
```

Here, the total cache size is assumed to be 16384 Bytes (16 KB) and block size as 4 Bytes.

```
bharat (master) Direct-Mapped-Cache-Simulation $ ./bin/sim wt traces/trace0.txt
CACHE HITS: 710738
CACHE MISSES: 30478
MEMORY READS: 30478
MEMORY WRITES: 238846
bharat (master) Direct-Mapped-Cache-Simulation $ ./bin/sim wb traces/trace0.txt

CACHE HITS: 710738
CACHE MISSES: 30478
MEMORY READS: 30478
MEMORY WRITES: 12539
bharat (master) Direct-Mapped-Cache-Simulation $ ./bin/sim wt traces/trace1.txt

CACHE HITS: 664
CACHE MISSES: 336
MEMORY READS: 336
MEMORY WRITES: 334
bharat (master) Direct-Mapped-Cache-Simulation $ ./bin/sim wb traces/trace1.txt

CACHE HITS: 664
CACHE MISSES: 336
MEMORY READS: 336
MEMORY WRITES: 0
bharat (master) Direct-Mapped-Cache-Simulation $ ./bin/sim wt traces/trace2.txt

CACHE HITS: 6725
CACHE MISSES: 3275
MEMORY READS: 3275
MEMORY WRITES: 2861
bharat (master) Direct-Mapped-Cache-Simulation $ ./bin/sim wb traces/trace2.txt

CACHE HITS: 6725
CACHE MISSES: 3275
MEMORY READS: 3275
MEMORY WRITES: 1204
```

Figure 10.1: Test run of Direct-Mapped Cache Simulator

10.2 Simulation of Our Proposed Cache

Simulator can be invoked by

```
python3 <program-name>
```

```
In [51]: runfile('C:/Users/ashwinss3/Desktop/Project/Cache_without_vc.py', wdir='C:/
Users/ashwinss3/Desktop/Project')
Misses in L1 : 1413
Hits in L1 : 114
Misses in L2 : 105
Hits in L2 : 1308
Time taken: 103965

In [52]: runfile('C:/Users/ashwinss3/Desktop/Project/Cache_with_vc.py', wdir='C:/Users/
ashwinss3/Desktop/Project')
Misses in L1 : 1413
Hits in L1 : 114
Misses in L2 : 105
Hits in L2 : 1266
Misses in Victim Cache : 1371
Hits in Victim Cache : 42
Time taken: 102824.7

In [53]: runfile('C:/Users/ashwinss3/Desktop/Project/Adaptive_cache_with_vc.py',
wdir='C:/Users/ashwinss3/Desktop/Project')
Misses in L1 : 1413
Hits in L1 : 114
Misses in L2 : 175
Hits in L2 : 1190
Misses in Victim Cache : 1365
Hits in Victim Cache : 48
Misses in case of Lru Cache: 161
Misses in case of fifo cache: 343
Time Taken: 120020.5
```

Figure 10.2: Test run of our proposed cache simulator

Chapter 11

Conclusion and Future Scope

11.1 Conclusion

In this projects we thoroughly analyzed the concept of Victim Cache and Adaptive Cache Replacement Algorithms. We designed a Cache Architecture that combines advantages of both these technologies and produces results with high hit rate, lower miss rate, lower hit time and thus higher cache performance.

11.2 Future Scope

By combining many interesting cache optimizations we can achieve a performance offered by all those optimizations combined, without much negotiating for costs or time. By doing so we can make our future caches even more fast, effective and at the same time can provide intelligence to it.

References

[1] Norman P. Jouppi, *Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers*, Digital Equipment Corporation Westem Research Lab, ISCA 1990.

[2] Ranjith Subramanian, Yannis Smaragdakis and Gabriel H. Loh *Adaptive Caches: Effective Shaping of Cache Behavior to Workloads*, Georgia Institute of Technology, Intel.

[3] Norman P. Jouppi, *Reducing Compulsory and Capacity Misses*, WRL Technical Note TN-53, Western Research Laboratory, August 1990.

[4] Introduction to Computer Architecture, Onur Mutlu, `http://www.ece.cmu.edu/~ece447/s15/doku.php`

[5] David A. Patterson, John L. Henessey, *Computer Organization And Design, Hardware/Software Interface, ARM Edition.*

[6] David A. Patterson, John L. Henessey, *Computer Architecture, A Quantative Approach, Fifth Edition.*

[7] Markus Kowarschik and Christian WeiB, *An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms, Deutsche Forschungsgemeinschaft, projects Ru 422/7-1,2,3, German Science Foundation, Germany.*

[8] Culler, David (1997), *Parallel Computer Architecture. Morgan Kaufmann Publishers.*

[9] *Reducing Miss Rate,* `http://ece-research.unm.edu/jimp/611/slides/chap5_3.html`

[10] *Larger Caches, Reducing Miss Rate,* `https://www.eecis.udel.edu/~sunshine/courses/F05/CIS662/class21.pdf`

[11] *Reducing Hit Time,* `http://ece-research.unm.edu/jimp/611/slides/chap5_4.html`

[12] *Memory Hierarchy - Reducing Hit Time and Main Memory,* https://people.eecs.berkeley.edu/~pattrsn/252F96/Lecture09.pdf

[13] *Memory Hierarchy - Five Ways to Reduce Miss Penalty,* http://bnrg.cs.berkeley.edu/~randy/Courses/CS252.S96/Lecture17.pdf