TABLE OF CONTENTS

ABSTRACT

Cache Memory is a special very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations. There are various different independent caches in a CPU, which store instructions and data.

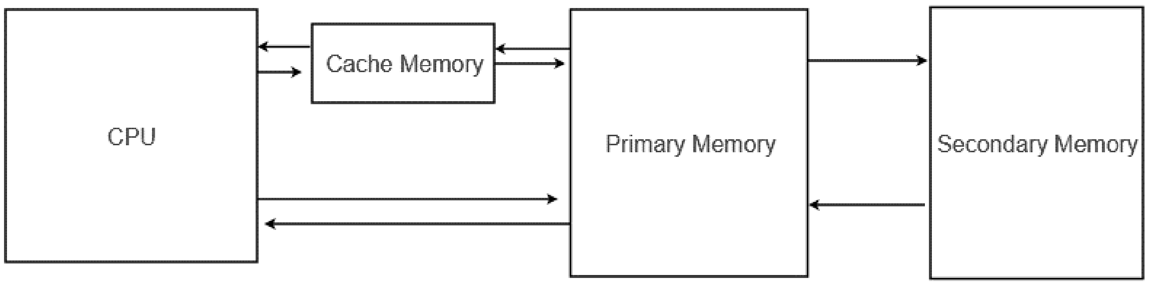


Fig. Levels of Memory

The data or contents of the main memory that are used frequently by CPU are stored in the cache memory so that the processor can easily access that data in a shorter time. Whenever the CPU needs to access memory, it first checks the cache memory. If the data is not found in cache memory, then the CPU moves into the main memory.

In this project, we aim to simulate the working of a cache memory, implementing different types of mapping, and hence, determine and analyse the performance of the cache memory by calculating the hit ratio, compulsory misses, capacity misses, and conflict misses.

Introduction

A cache simulator is a software tool that mimics the behaviour of a hardware cache subsystem. A cache is used to reduce the average cost of accessing main memory from the processor. The performance difference between main memory and the processor has increased over time, necessitating the development of more sophisticated caches. One way to evaluate a new cache subsystem is by producing a hardware prototype and evaluating its performance. However, this process is often costly and time-consuming.

An alternative way is to implement the new idea in a simulator and evaluate it using simulation. This facilitates design space exploration and makes it accessible to a wider community of researchers and students. Simulators can also be used to study the behaviour of programs on existing processors where hardware performance events are not sufficient or unavailable. In addition, when the study needs to be performed on many different existing systems, one could avoid purchasing all these systems by using a simulator.

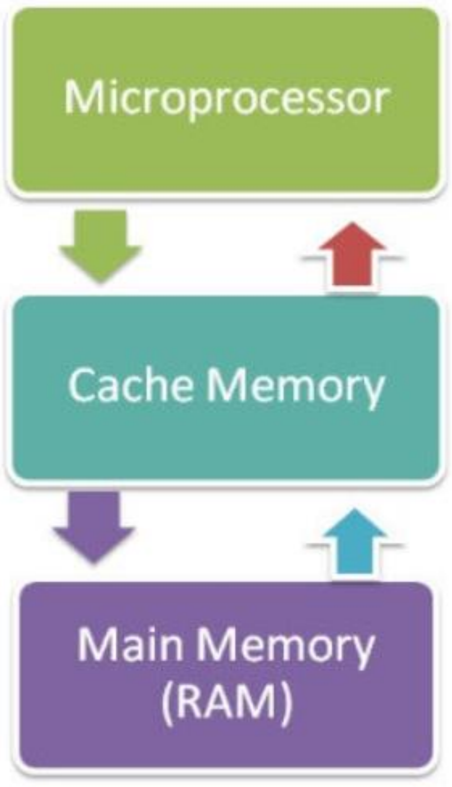
The scope of this project is limited to traditional CPU cache. Hardware caches have been used in almost all kinds of processors and appear in many different designs and configurations. Modern compute-capable GPUs typically have a two-level cache hierarchy. In contrast to traditional multicore CPUs, GPUs may have separate caches for scalar and vector data and may have special caches such as constant and texture caches. Some GPUs oer caches with configurable sizes

Cache Memory

Cache memory is one of the fastest memories inside a computer which acts as a buffer or mediator between CPU and Memory (RAM). When CPU requires some data element it goes to Cache and it that data element is present in cache, it fetches it; otherwise, cache controller requests the data from memory. Cache contains most frequently accessed data locations. While giving high speed of data access, cache is equally expensive as compared to other memories in a machine.

Major purpose of a Cache is to reduce the memory access time because going to primary memory costs a lot more time compared to cache. With the development of high-speed processors, memory access has been a bottleneck for the throughput of computational machines for decades. Multiple advances have been carried out to improve the throughput of computers, one of which was the introduction of cache memory.

There are two main parts of the cache one of which is **Directory** which stores the addresses of the lines and the other one is **Data line** which holds the data that is stored in the cache memory which is further addressed by the directory.

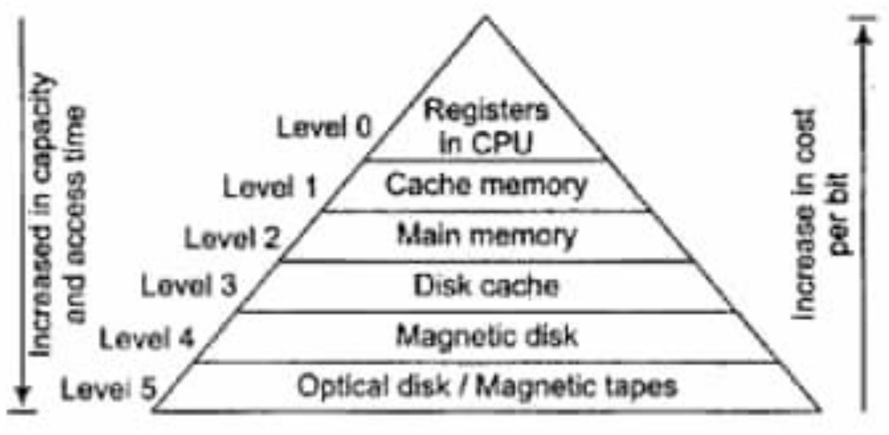


Memory Hierarchy

The memory hierarchy separates computer storage into a hierarchy based on response time. Since response time, complexity, and capacity are related, the levels may also be distinguished by their performance and controlling technologies. Memory hierarchy affects performance in computer architectural design, algorithm predictions, and lower-level programming constructs involving locality of reference.

A computer has different types of memories which serve different purpose depending upon their speed and cost. Some of these memories are volatile which means that they lose their states when power is turned off, while others are non-volatile which retain their states even when the power is turned off. However, main purpose of all these memories is to store data and provide it to the processing unit when required.

To reduce the latency of data transfer between memories and processing units, multiple strategies and techniques have been adopted over the decades. Major classification of memory includes, primary memory and secondary memory. Primary Memory includes the internal memories of a CPU which usually regards to registers, cache and RAM, whereas secondary memory includes hard disks or compact disks etc.



Cache Controller

Cache controller handles the data requests and controls data transfer between Cache and Processor & Cache and Memory. When processor requests a data element, cache controller checks for that element in Cache, and provides it to processor if present. In case, the required data element is not present in cache, the cache controller requests that data from memory. The read and write requests to memory are handled by the memory controller. Cache controllers handle the request by dividing it into three parts tag, set index and data index. Set index is used to locate the corresponding line of the cache memory. If the valid bit represents that the line is active, tag is compared. In case of success in both these cases, the element is fetched and it is considered as Cache Hit, otherwise, it is a Cache Miss.

Cache Hit

If the data requested by the processor is present in the cache, it is accessed and provided to processor by cache controller and this is regarded as Cache Hit. Throughput of a system largely depends upon the frequency of cache hits because in this case, Processor has to face a very short latency.

Cache Miss

If the data requested by the processor is not present in the cache, it is then requested from the memory and brought into the cache to make it available for the processor and this is regarded as Cache Miss. In this case, the data request is delayed by a reasonable amount of time and the processor has to wait for the data to arrive from the memory which is a comparatively time taking process.

Processor performance increase due to cache hierarchy depends on the number of accesses to the cache that satisfy block requests from the cache (cache hits) versus those that do not.

Unsuccessful attempts to read or write data from the cache (cache misses) result in lower level or main memory access, which increases latency. There are three basic types of cache misses known as the **3Cs**.

Compulsory Misses

Data that has never been accessed resides only in the Main Memory. When processor receives request for this data, it will cause a Cache Miss, as the data was never loaded into the Cache. These kinds of unavoidable misses are called Compulsory Misses. They are also known as Cold-Start Misses or First Reference Misses.

Compulsory Misses can be reduced by increasing the block size. Increasing the block size allows more data to be fetched into the Cache. This improves the chances of a new memory location request already being present in the Cache. However, loading a bigger block will take more time. Thus, this process can have a negative effect on performance by increasing miss penalty. Prefetching can also reduce Compulsory Misses.

Capacity Misses

Capacity misses occur when the Cache cannot contain all the blocks that are required during program execution. During execution, the blocks need to be evicted from the Cache to make room for other blocks. The block is later fetched back into the Cache when it is needed again. Thus, it is the limit on the number of blocks a Cache can hold that causes these kinds of misses.

Capacity Misses can be reduced by increasing the size of the Cache. This gives room for a greater number of blocks. However, this process increases the access time which has a negative effect on overall performance.

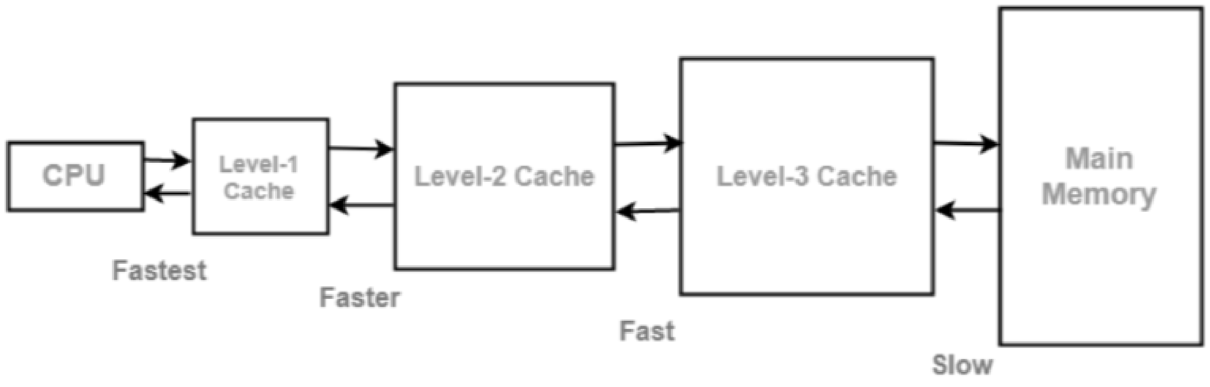
Conflict Misses

Conflict Misses are a variant of Capacity misses where multiple blocks compete for the same set. Thus, they occur only in direct-mapped or set-associative Caches. Fully associative caches have no Conflict Misses. They are also known as Collision Misses.

Increasing the associativity will reduce Conflict Misses. However, this process increases the access time which has a negative effect on overall performance.

Levels of Cache

There are mainly three levels of cache (L1, L2, L3) which are categorized based on their speed and capacity. Going from L1 to L3, memory access time and storage capacity increases.



L1

L1 (Level 1) cache is the fastest memory that is present in a computer system. In terms of priority of access, the L1 cache has the data the CPU is most likely to need while completing a certain task.

The size of the L1 cache depends on the CPU. Usually, its size is up to 256KB, however, some top-end consumer CPUs now feature a 1MB L1 cache, like the Intel i9-9980XE, but these cost a huge amount of money. Some server chipsets, like Intel's Xeon range, also feature a 1-2MB L1 memory cache.

There is no "standard" L1 cache size, so you must check the CPU specs to determine the exact L1 memory cache size before purchasing.

The L1 cache is usually split into two sections: the instruction cache and the data cache. The instruction cache deals with the information about the operation that the CPU must perform, while the data cache holds the data on which the operation is to be performed.

L2

L2 (Level 2) cache is slower than the L1 cache but bigger in size. Where an L1 cache may measure in kilobytes, modern L2 memory caches measure in megabytes. For example, AMD's highly rated Ryzen 5 5600X has a 384KB L1 cache and a 3MB L2 cache (plus a 32MB L3 cache).

The L2 cache size varies depending on the CPU, but its size is typically between 256KB to 8MB. Most modern CPUs will pack more than a 256KB L2 cache, and this size is now considered small. Furthermore, some of the most powerful modern CPUs have a larger L2 memory cache, exceeding 8MB.

When it comes to speed, the L2 cache lags behind the L1 cache but is still much faster than your system RAM. The L1 memory cache is typically 100 times faster than your RAM, while the L2 cache is around 25 times faster.

L3

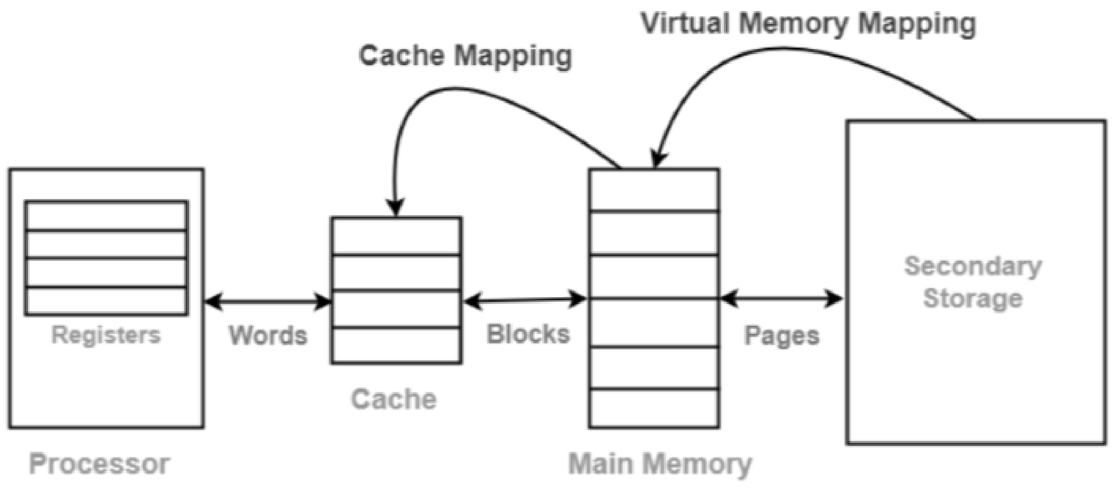
Onto the L3 (Level 3) cache. In the early days, the L3 memory cache was actually found on the motherboard. This was a very long time ago, back when most CPUs were just single-core processors. Now, the L3 cache in your CPU can be massive, with top-end consumer CPUs featuring L3 caches up to 32MB. Some server CPU L3 caches can exceed this, featuring up to 64MB.

The L3 cache is the largest but also the slowest cache memory unit. Modern CPUs include the L3 cache on the CPU itself. But while the L1 and L2 cache exist for each core on the chip itself, the L3 cache is more akin to a general memory pool that the entire chip can make use of.

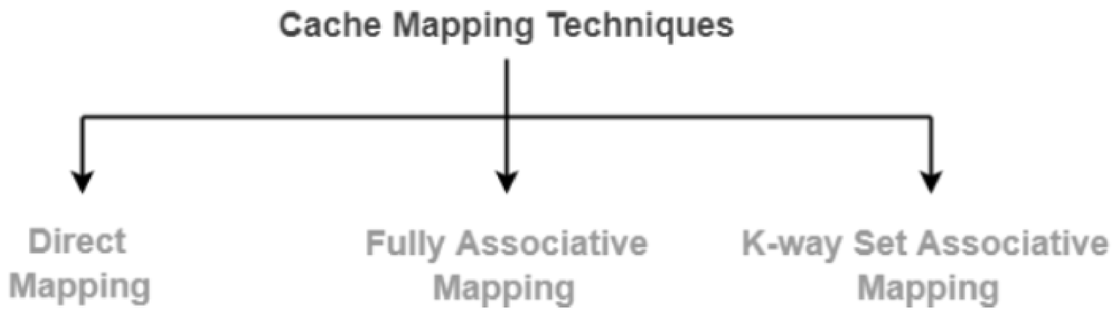
Cache Mappings

Cache mapping is a technique by which the contents of main memory are brought into the cache memory.

The following diagram illustrates the mapping process-



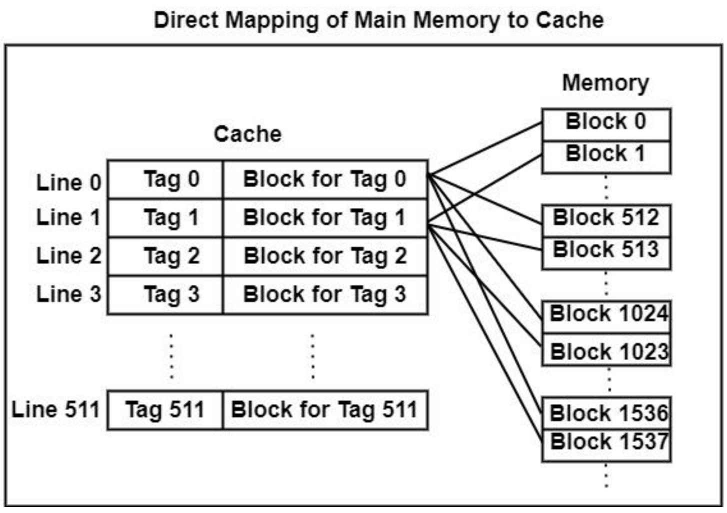
Cache Mapping can further be divided into the following techniques: -



DIRECT CACHE MAPPING

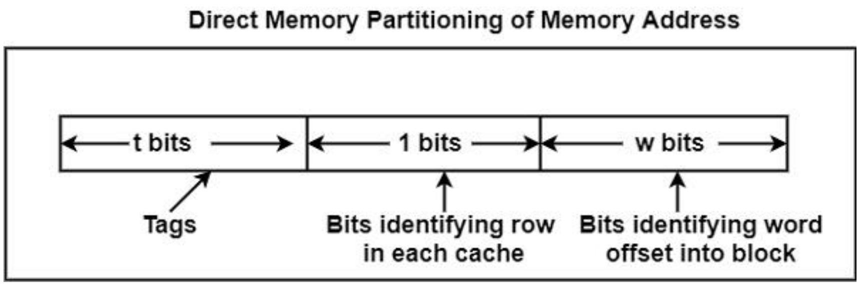
Direct mapping is a procedure used to assign each memory block in the main memory to a particular line in the cache. If a line is already filled with a memory block and a new block needs to be loaded, then the old block is discarded from the cache.

The figure shows how multiple blocks from the example are mapped to each line in the cache.



Just like locating a word within a block, bits are taken from the main memory address to uniquely describe the line in the cache where a block can be stored.

Direct mapping divides an address into three parts: t tag bits, l line bits, and w word bits. The word bits are the least significant bits that identify the specific word within a block of memory.

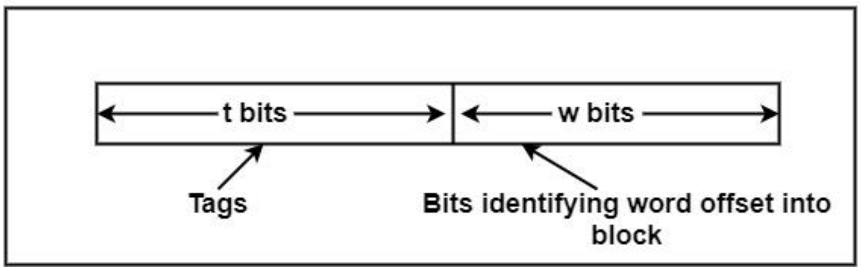

Direct Memory Partitioning of Memory Address
t bits
1 bits
w bits
Tags
Bits identifying row in each cache
Bits identifying word offset into block

The line bits are the next least significant bits that identify the line of the cache in which the block is stored. The remaining bits are stored along with the block as the tag which locates the block’s position in the main memory.

FULLY ASSOCIATIVE CACHE MAPPING

In the associative mapping function, any block of main memory can probably consist of any cache block position. It breaks the main memory address into two parts - the word ID and a tag as shown in the figure. To check for a block stored in the memory, the tag is pulled from the memory address and a search is performed through all of the lines of the cache to see if the block is present.

Associative Partitioning of Memory Address



This method of searching for a block within a cache appears like it might be a slow process, but it is not. Each line of the cache has its compare circuitry, which can quickly analyse whether or not the block is contained at that line. With all of the lines performing this comparison process in parallel, the correct line is identified quickly.

This mapping technique is designed to solve a problem that exists with direct mapping where two active blocks of memory could map to the same line of the cache. When this happens, neither block of memory is allowed to stay in the cache as it is replaced quickly by the competing block. This leads to a condition that is referred to as thrashing.

In thrashing, a line in the cache goes back and forth between two or more blocks, usually replacing a block even before the processor goes through it. Thrashing can be avoided by allowing a block of memory to map to any line of the cache.

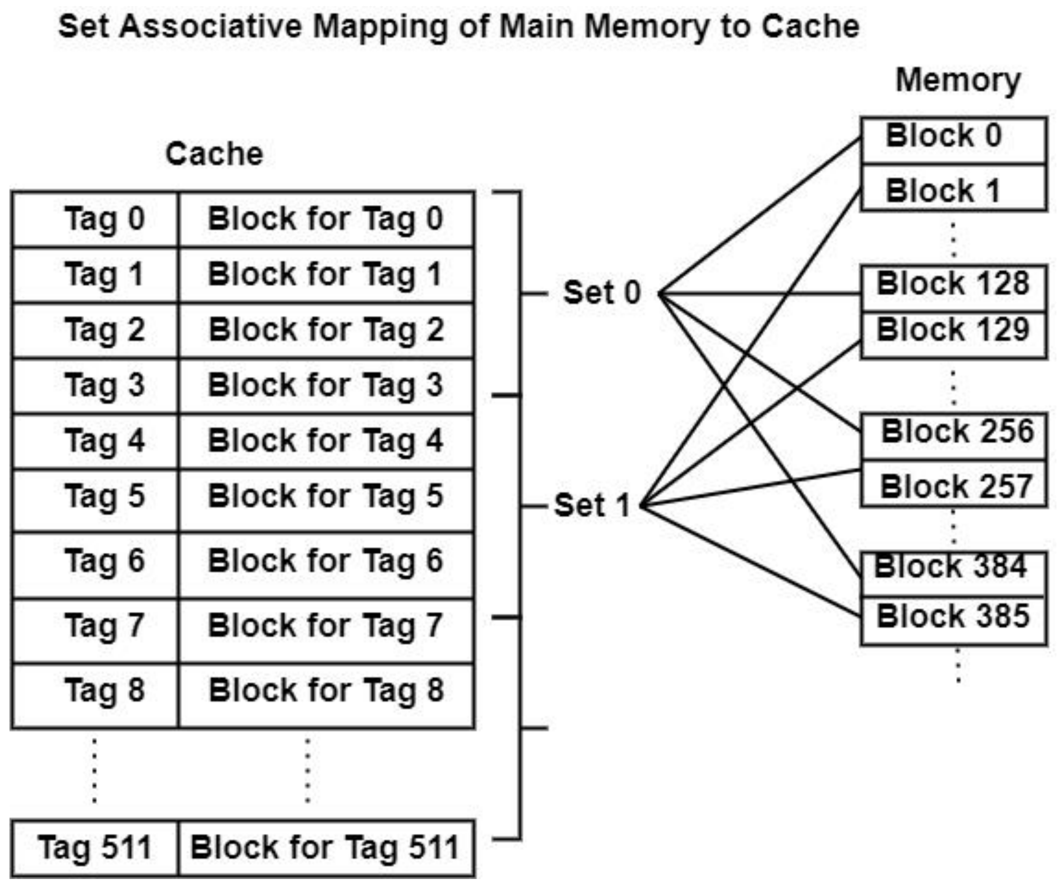
There are many replacement algorithms, none of which has any precedence over the others. In an attempt to realize the fastest operation, each of these algorithms is implemented in hardware.

- **Least Recently Used (LRU)** − This approach restores the block that has not been read by the processor in the highest period.
- **First In First Out (FIFO)** − This approach restores the block that has been in cache the highest.
- **Least Frequently Used (LFU)** − This approach restores the block which has fewer hits because being loaded into the cache.
- **Random** − This approach randomly chooses a block to be replaced. Its execution is slightly lower than LRU, FIFO, or LFU.

SET ASSOCIATIVE CACHE MAPPING

Set associative mapping combines direct mapping with fully associative mapping by arrangement lines of a cache into sets. The sets are persistent using a direct mapping scheme. However, the lines within each set are treated as a small fully associative cache where any block that can save in the set can be stored to any line inside the set.

The diagram represents this arrangement using a sample cache that uses four lines to a set.



A set-associative cache that includes k lines per set is known as a k way set-associative cache. Because the mapping approach uses the memory address only like direct mapping does, the number of lines included in a set should be similar to an integer power of two, for example, two, four, eight, sixteen, etc.

Measuring Cache Performance

AMAT (Average Memory Access Time) is a measure that is used to test performance of a memory. It is calculated using following formula:

$$\boldsymbol{AMAT = Hit\,Time + Miss\,Rate\ \times Miss\,Penalty}$$

Performance of a cache can be improved by following things:

1. Reducing Miss Rate

2. Reducing Miss Penalty

3. Reducing Time to Hit

At the start, there is compulsory miss because it is the first access by the processor to the memory. Due to limited size of the cache, data that was used in the past and not required now is replaced with the data that is required now and in that case cache miss occurs. There can be misses due to collision too. Reducingthe number of misses will increase the number of Hits which eventually increases the throughput by increasing memory access time significantly.

Code Implementation

Set Associative Mapping

```
index_addr=index_addr * type;
       for (int i=0; i < type ; i++)
       {
           if (cache[0][index_addr+i]==tag_addr)
           {
               return 1;
           }
       }
       for (int j=0; j < type; j++)
       {
           if (cache[1][index_addr+j] == -1)
           {
               compulsory_misses++;
               cache[0][index_addr+j]=tag_addr;
               cache[1][index_addr+j]=1;
               return 0;
           }
       }

       srand(time(NULL));
       int x=rand()%(type);
       cache[0][index_addr+x]=tag_addr;
       cache[1][index_addr+x]=1;
       capacity_misses++;
       return 0;
```

Fully Associative Mapping

```
if (type==0)    // LRU /////////
       {
           if (block_counter < number_of_blocks)
           {
               for (int i=0; i < number_of_blocks; i++)
               {
                   if (cache[0][i]==addr >> shift_offset)
                   {
                       detected=true;
                       cache[1][i]=block_counter;
                       block_counter--;
                       return detected; //hit
                   }
```

```
                }

                if (!detected)
                {
                    compulsory_misses++;
                    cache[0][block_counter]=addr>>shift_offset;
                    cache[1][block_counter]=block_counter;
                    return false;  //miss
                }
            }
            else  // block counter is more than block size
            {
                //check for existence
                for (int i=0; i < number_of_blocks; i++)
                {
                    if (cache[0][i]==(addr >> shift_offset))
                    {
                        detected=true;
                        cache[1][i]=block_counter;
                        //block_counter--;
                        return detected; //hit
                    }
                }

                if (!detected)
                {
                    int compare=0;
                    for (int i=1; i < number_of_blocks; i++)
                    {
                        if (cache[1][compare] > cache[1][i])
                            compare=i;
                    }
                    cache[0][compare]=addr >> shift_offset;
                    cache[1][compare]=block_counter;
                    capacity_misses++;
                    return false; //miss

                }
            }
        }   // end of LRU

        else if (type==1)   // LFU ///////////////
        {
            if (block_counter < number_of_blocks)
            {
                for (int i=0; i < number_of_blocks; i++)
                {
                    if (cache[0][i]==addr >> shift_offset)
                    {
                        detected=true;
                        cache[1][i]=cache[1][i]+1;
                        block_counter--;
                        return detected; //hit
                    }
                }

                if (!detected)
                {
                    cache[0][block_counter]=addr>>shift_offset;
                    cache[1][block_counter]=-1;
                    compulsory_misses++;
                    return false;  //miss
                }
            }
            else  // block counter is more than block size
            {
                //check for existence
                for (int i=0; i < number_of_blocks; i++)
                {
                    if (cache[0][i]==(addr >> shift_offset))
                    {
                        detected=true;
                        cache[1][i]++;
                        block_counter--;
                        return detected; //hit
                    }
                }
                if (!detected)
                {
                    int compare2=0;
                    for (int i=1; i < number_of_blocks; i++)
                    {
                        if (cache[1][compare2] >= cache[1][i])
                            compare2=i;
                    }
                    cache[0][compare2]=addr >> shift_offset;
                    cache[1][compare2]=-1;
                    capacity_misses++;
                    return false; //miss
                }
            }

        }  // end if LFU

        else if (type==2)
        {
            if (block_counter < number_of_blocks)
            {
                for (int i=0; i < number_of_blocks; i++)
                {
                    if (cache[0][i]==addr >> shift_offset)
                    {
                        detected=true;
                        block_counter--;
                        return detected; //hit
                    }
                }

                if (!detected)
                {
                    cache[0][block_counter]=addr>>shift_offset;
                    cache[1][block_counter]=block_counter;
                    compulsory_misses++;
                    return false;  //miss
                }
            }
            else  // block counter is more than block size
            {
                //check for existence
                for (int i=0; i < number_of_blocks; i++)
                {
                    if (cache[0][i]==(addr >> shift_offset))
                    {
                        detected=true;
                        //block_counter--;
                        return detected; //hit
                    }
                }

                if (!detected)
                {
                    int compare=0;
                    for (int i=1; i < number_of_blocks; i++)
                    {
                        if (cache[1][compare] > cache[1][i])
                            compare=i;
                    }
                    cache[0][compare]=addr >> shift_offset;
                    cache[1][compare]=block_counter;
                    capacity_misses++;
                    return false; //miss
                }
            }
        }// end of FIFO
        else if (type==3)
        {
            if (block_counter < number_of_blocks)
            {
                for (int i=0; i < number_of_blocks; i++)
                {
                    if (cache[0][i]==addr >> shift_offset)
                    {
                        detected=true;
                        block_counter--;
                        return detected; //hit
                    }
                }

                if (!detected)
                {
                    cache[0][block_counter]=addr>>shift_offset;
                    compulsory_misses++;
                    return false;  //miss
                }
            }
            else  // block counter is more than block size
            {
                //check for existence
                for (int i=0; i < number_of_blocks; i++)
                {
                    if (cache[0][i]==(addr >> shift_offset))
                    {
                        detected=true;

                        return detected; //hit
                    }
                }

                if (!detected)
                {
                    srand(time(NULL));
                    cache[0][rand()%number_of_blocks]=addr >> shift_offset;
                    capacity_misses++;
                    return 0; //miss
                }
            }
        }
    }
```

Direct Mapping

```
if (cache[0][index_addr]==tag_addr)
        {
            return true;
        }
        else
            {
                cache[0][index_addr]= tag_addr;
                for (int i=0; i < number_of_blocks; i++)
                {
                    if (cache[1][i]!=1)
                    {   misses_flag=false;
                        i=number_of_blocks;}

                }
                //calculating misses
                if (misses_flag)
                    capacity_misses++;   // Capacity miss because the cache is
full
                else
                {
                    if(cache[1][index_addr]==1)
                        conflict_misses++;
                    else
                    {
                        compulsory_misses++;
                    }
                }
                cache[1][index_addr]= 1;
                return 0;
            }
    }
```

RESULTS

For **direct mapped cache**, with a **constant cache size of 64kB**, results obtained based on generating **1 million address per simulation** are as follows

Block Size	Hits	Compulsory M	Capacity M	Conflict M
4 bytes	750,000	16,383	233,616	0
8 bytes	875,000	8,192	116,808	0
16 bytes	937,500	4,096	58,404	0
32 bytes	968,750	2,048	29,202	0
64 bytes	984,375	1,024	14,601	0
128 bytes	992,187	512	7,301	0