INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR

CS 215 Semester IV
Course Project

CACHE SIMULATOR

By:
MIHIR SUTARIYA (20110208)
SANDEEP DESAI (20110052)
DHEERAJ YADAV (20110056)
RAHUL RAI (20110156)

Contents

1 Abstract

For many developers/software engineers, the processor's cache is seen as a black box that increases the program execution speed. The tools used to analyze the cache performance of the processor are known as cache simulators. These tools simulate the cache interactions of the running program and then provide us with feedback and tell us where to improve. The simulation can be done by specifying the cache parameters like the Block size, associativity, cache size, and the replacement policy that is least recently used, Last in first out (LIFO) and First in First Out (FIFO), and the trace of the program file. The cache simulator will output the type of cache used (Fully associative/set-associative/direct-mapped), the number of read and write misses /access, and the compulsory, capacity, and conflict miss. We will compare our result obtained to the result of a cache simulator taken from the web and derive the conclusion for the same.

2 Introduction

The CPU is the core of the computer and is responsible for executing instructions, handling memory and sending signals. In between the CPU and the main memory a modern hardware called as CPU cache is present often called as cache. The cache acts as a temporary storage and it reduces the time for the read and write operations. Accessing the data from the cache is much faster than accessing the data from the main memory.

The project's primary goal is to create a cache simulator and check if it correctly outputs the various cache parameters. We will be simulating the memory read and write operations in a virtual cache. Given a sequence of instructions labeled as read or write, the memory data can be stored and read from a virtually created cache. The virtual cache must be instantiated according to the given cache parameters, such as size, block size, and associativity, along with the replacement policy to follow for resolving conflicts. The simulator takes these memory instructions one by one and performs the above-instantiated cache operations. The goal is to record the number of hits and misses and the number of conflicts according to the Replacement policy given.

Through this project, we will get an in depth understanding of the various cache parameters and its various operations. The project is of great importance for the future computer architecture courses to teach about the basic cache principles and how the whole simulator works.The simulation results are useful in providing us feedback and telling us where to improve. We will be able to evaluate and enhance the cache performance using the Cache Simulator.

3 Literature Review

British computer scientist Maurice Wikes first introduced the cache in April 1965. He named cache as slave memory. In his paper, he mentioned that slave mermories have recently come into prominence to reduce instruction access in otherwise conventional computers. Slave memory will automatically accumulate and keep it for future use to reduce the penalty. When we have a processor without a cache, we need much more cycles of a clock to access the memory. However, when we use intermediate hardware close to the processor and use the data rather than going back to memory again will increase performance of the CPU. When there was no cache concept, the CPU time was mostly dependent on the number of memory access instructions, so Maurice Wikes proposed a cache.

In this paper, the proposed cache was 1 level, and also it was not split for data and instruction. IBM 801 CPU (1976) has introduced a split cache with two parts, one for instructions and one for data. After that, they also introduced multilevel caching. There are usually three caching L1, L2, and L3. L2 and L3 are not generally splitted. After L1, caches may have been shared between multiple cores.

Cache performance is dependent on many parameters like the cache replacement policy. For faster access, the capacity of the cache is low. So if we access new data which is not in the cache, we need to find that data in the memory and then replace that data in the cache. The priority in which we chose the data is the cache replacement policy. Cache replacement policy is also an optimization problem limited not just to cache and can be applied in these algorithms to various problems.

Its application is not just a cache but these algorithms can be applied to other problems. There are many replacement policies such as random replacement, first in, first out, last in, first out, least recently used, most recently used, etc. Other parameters are block size, cache capacity, and associativity. Its values are dependent on different applications.

First, we have seen the splitting of cache. A split cache is much better because instruction access and data access from memory has different locality properties. When a cache is split, we can define its parameters based on its type. However, splitting also increases hardware implementation complexity.

Multilevel will increase the capacity because after level 1, we can increase its capacity, reducing the miss penalty because it is possible that when we did not find data in the first level, we would find it in higher levels.
Replacement policy is dependent on sequence and patterns of memory access. So, it is not randomized. We have an opportunity to increase the number of hits based on these sequences and patterns. Generally, LRU policy is better than FIFO, LIFO, RR.
Our proposed simulation of the cache implements all the above features, and also we can characterize it by parameters and compare the performances. Here we are not taking splitting of the cache and also different levels of the cache, but the design is such that it is easy to extend it further to a multilevel cache.

4 Project Idea

4.1 In Depth Problem Statement

For simulating a cache virtually, we first need to create a suitable structure for the cache. We need to choose which kind of data structure we need to use; for instance, in C++, we can define a struct for suitable applications, vectors, and lists from the STL library of C++ for the dynamic storing of data. Based on the given input, we will be able to identify the type of mapping the cache uses. Thus, there is a need to code to dynamically simulate cache having any mapping: direct mapping/ set-associative mapping/ fully associative mapping.

The simulator will take user input for the replacement policy and the trace file and simulate the program for the three replacement policies: Least Recently Used, First In First Out (FIFO), and Last In First Out (LIFO).

Our simulator should calculate the number of cache access, read/write misses, compulsory misses, capacity misses, and read/write access; thus, our code should keep track of all of the data. Further, to analyze data produced by our simulator and compare it with standard cache simulators available in the market, we must keep track of hit rates and the number of misses. Here we are only implementing a single-level cache simulator. Still, in the future, if we want to upgrade it to implement a multi-level cache, we should keep track of read/write misses separately so that we can calculate read/write penalties.

4.2 High Level Overview of Solution

Replacement Policies:

High Level code for LRU policy:

```
Find block(current address)
If the block is missing and cache is occupied fully => create
space
{
Remove the least recently used block
Add the block accessed at the end
}
```

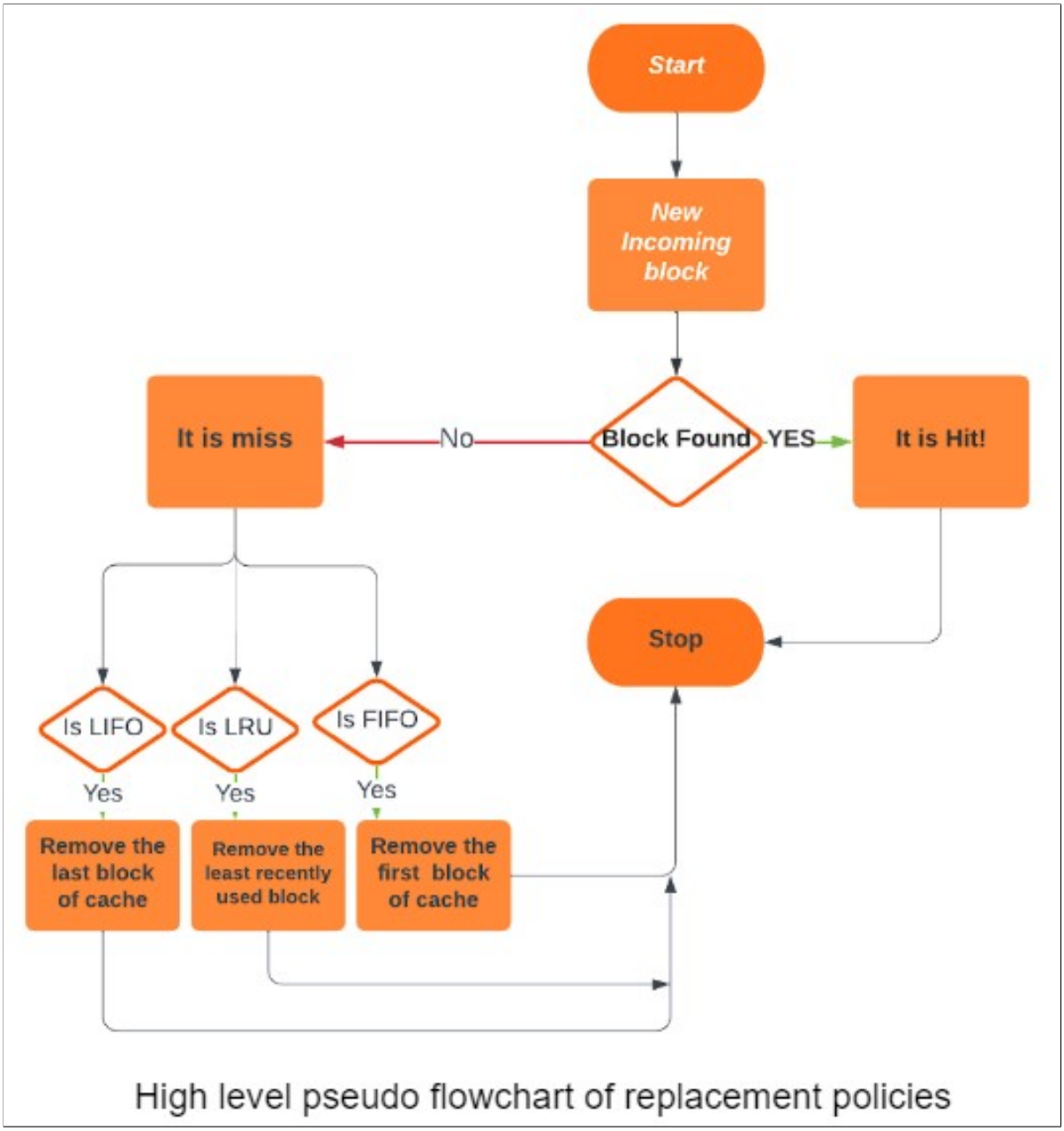
High level pseudo code for FIFO policy:

```
Find block(current address)
If the block is missing and cache is occupied fully => create
space
{
Remove the first block
Add the block accessed at the end
}
```

High level pseudo code for LIFO policy:

```
Find block(current address)
If the block is missing and cache is occupied fully => create
space
{
Remove the last block
Add the block accessed at the end
}
```

The following flowchart shows the high level overview of simulating various replacement policies:



High level pseudo flowchart of replacement policies

4.3 Calculation of Misses/Hits

4.3.1 Conflict Misses

It will occur when a set is full and the address is not present in the set.

4.3.2 Compulsory Misses

It will occur when the required address is not present in the set and the set is also not full.

4.3.3 Capacity Misses

It will occur when all the sets of the cache are full and the required memory address is not present in the set.

4.3.4 Hits

$$\text{Total Hits} = \text{Number of Memory access - Number of misses}$$

4.3.5 Hit Rate

$$\text{Hit Rate} = \frac{\text{Number of hits}}{\text{Number of Memory access}} \times 100$$

4.4 Structure of Cache:

Each cache will have some number of sets, each set will contain a number of blocks equal to associativity. Each block will consist of tag bits, index bits and block offset.

```
blocks = capacity/block_size;
No of sets = No of blocks/associativity;
offset = log(block_size)/log(2);
index = log(No of sets)/log(2);
tag = 32 - offset - index;
```

Note that here we have assumed that each block in memory has an address of size 32 bits.

5 Project Implementation

5.1 Input/Output

We have implemented a cache simulator in C++. The C++ program take the following command line arguments that is:

- Block size in bytes
- Associativity of the cache
- Cache size in bytes
- Replacement policy which takes 0, 1, 2 as the inputs. For 0 we have implemented the LRU cache, for 1 FIFO is implemented and for 2 LIFO is implemented.

On compilation with the above command line parameters and trace file containing the memory addresses will result in the following output:

- Type of cache(Fully associative/set-associative/direct mapped)
- Replacement policy
- Number of cache access
- Number of read/write misses/hits
- Number of compulsory misses
- Number of capacity misses
- Number of conflict misses
- Number of read/write access

5.2 Cache Structure

The cache is instantiated using the vector of lists containing struct objects. This overall structure of the cache implemented in the code is shown in the below figure. The code snippet shows the struct declared for each block in the cache.

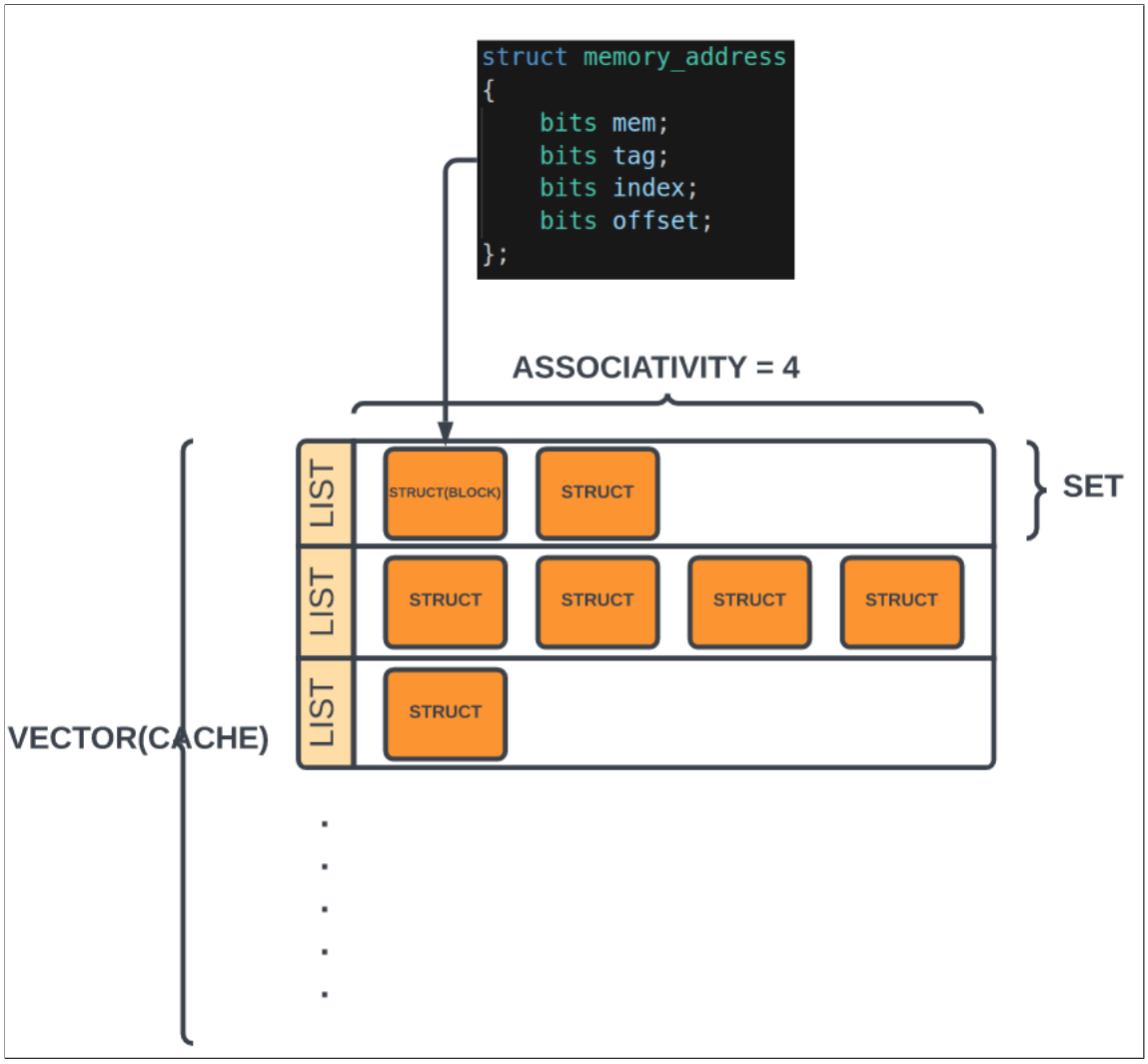


Figure 1: *Catch Structure*

5.3 Cache Simulation

Cache simulation operations are to be performed based on the state of the cache. The flow of the cache process in the code is depicted in the following flowchart. Based on the states and finding the operation of the tag, only 5 end conditions are possible. These 5 operations vary based on the replacement policy used.

NOTE:

- In the following flowcharts, **cache full** means the set corresponding to a particular index is full.
- At the same time, a **new cache** means that none of the blocks of all the sets is occupied.

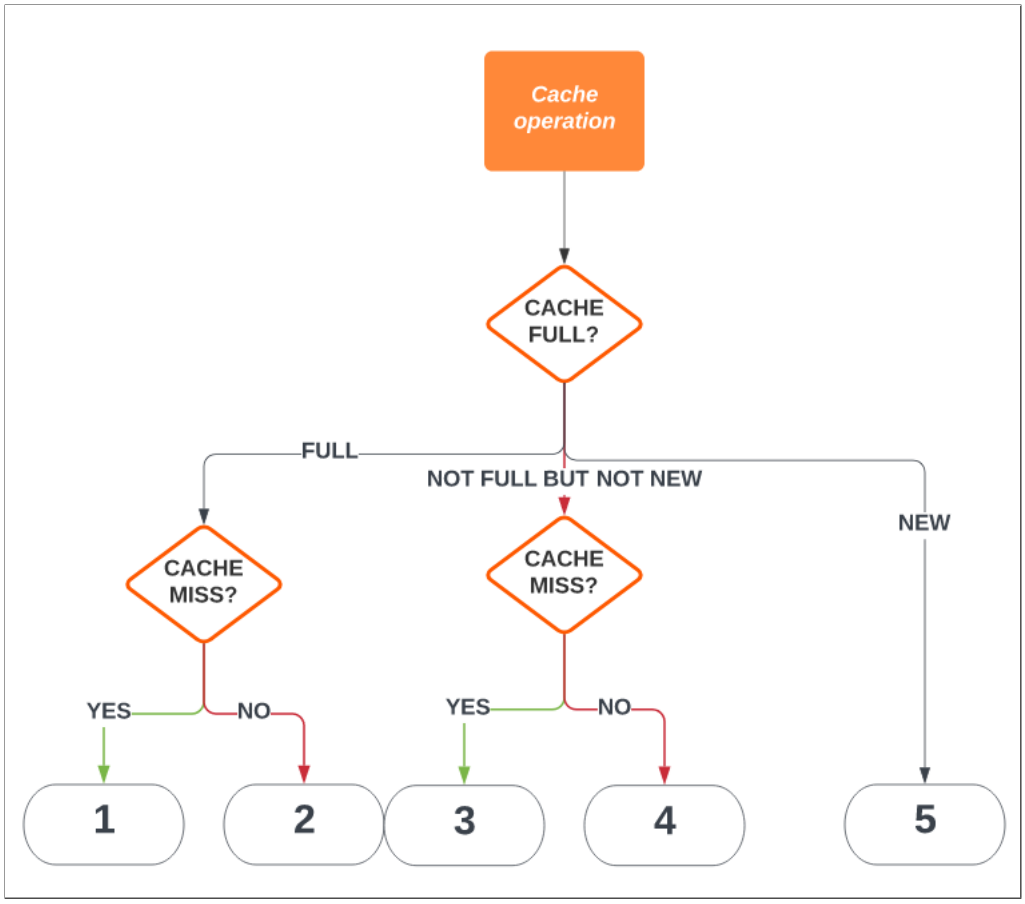


Figure 2: *Flowchart for Cache Simulation*

5.3.1 Least Recently Used

The above-mentioned 5 operations for LRU replacement policy are shown below. The code implementation LRU is shown in the below figure.

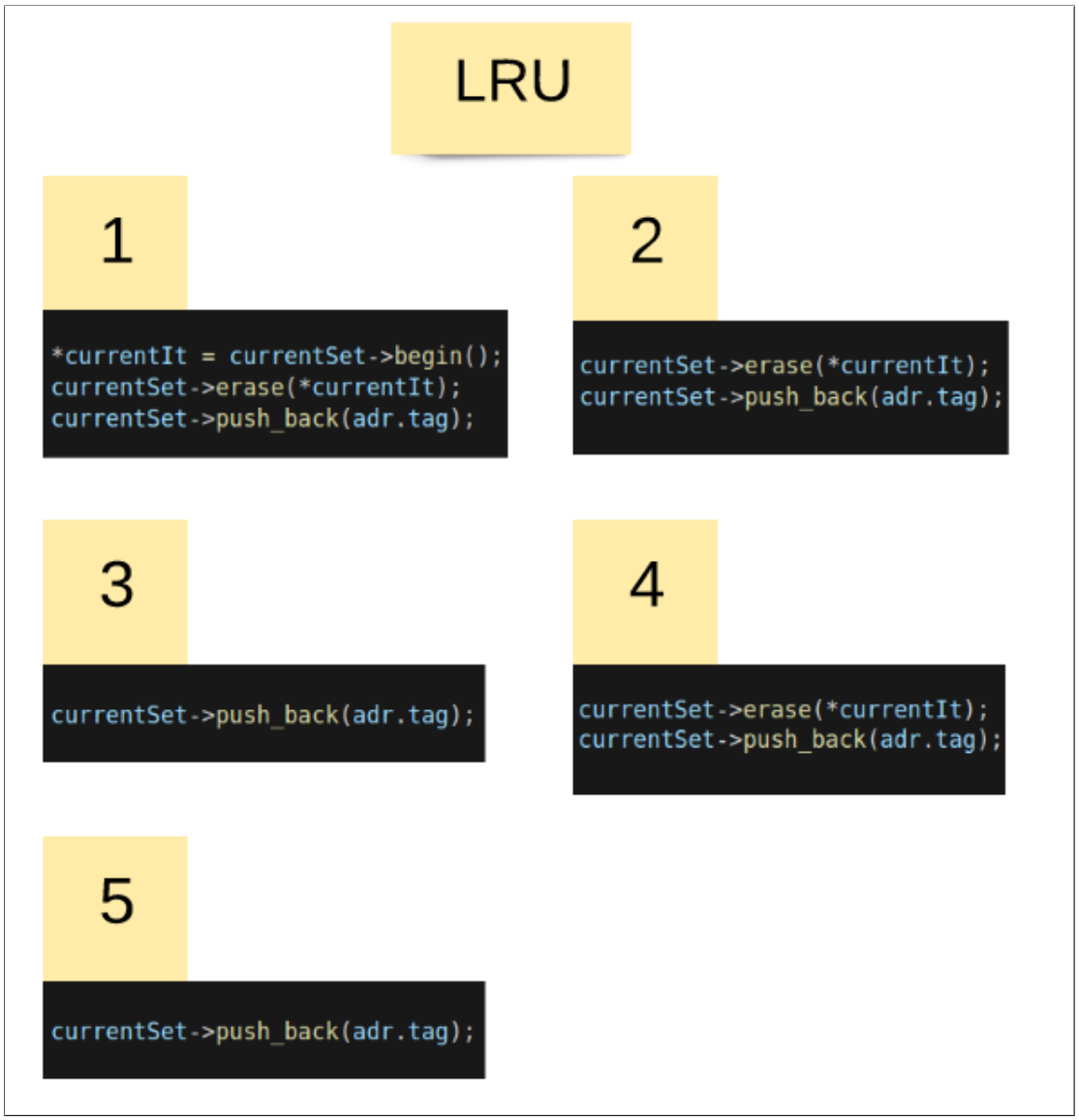


Figure 3: *Code for the least recently Used (LRU)*

5.3.2 First In First Out(FIFO)

The above-mentioned 5 operations for FIFO replacement policy are shown below. The code implementation LRU is shown in the below figure.

Figure 4: *Code for FIFO*

5.3.3 Last In First Out(LIFO)

The above-mentioned 5 operations for LIFO replacement policy are shown below. The code implementation LRU is shown in the below figure.

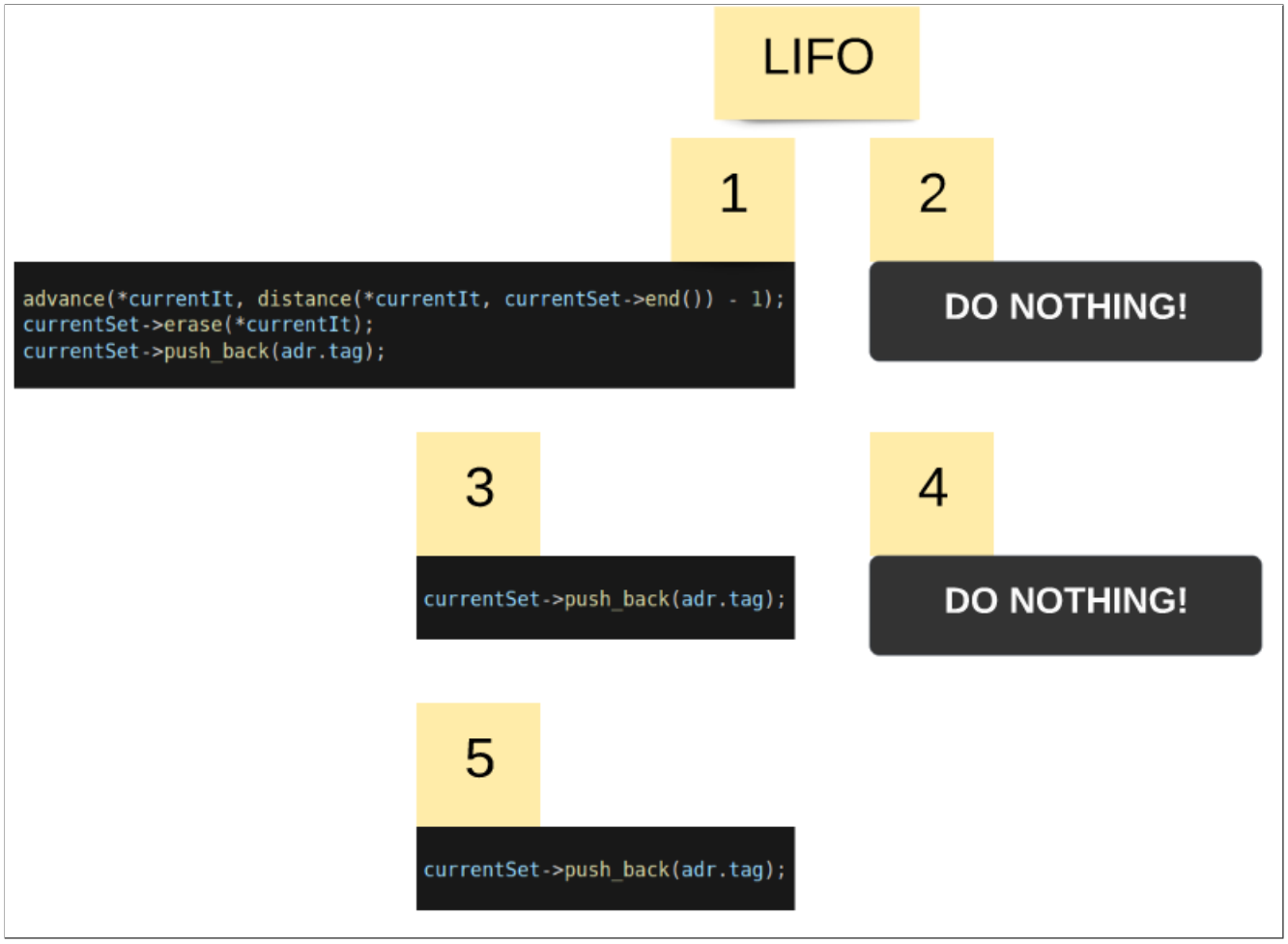


Figure 5: *Code for LIFO*

5.4 Hits and Misses

The misses and hits are also calculated in the 5 states of the simulation function. The type of miss also depends on the state. In the implementation, these three misses(compulsory miss, conflict miss and capacity) and hits are calculated along with the operations of replacement policy.

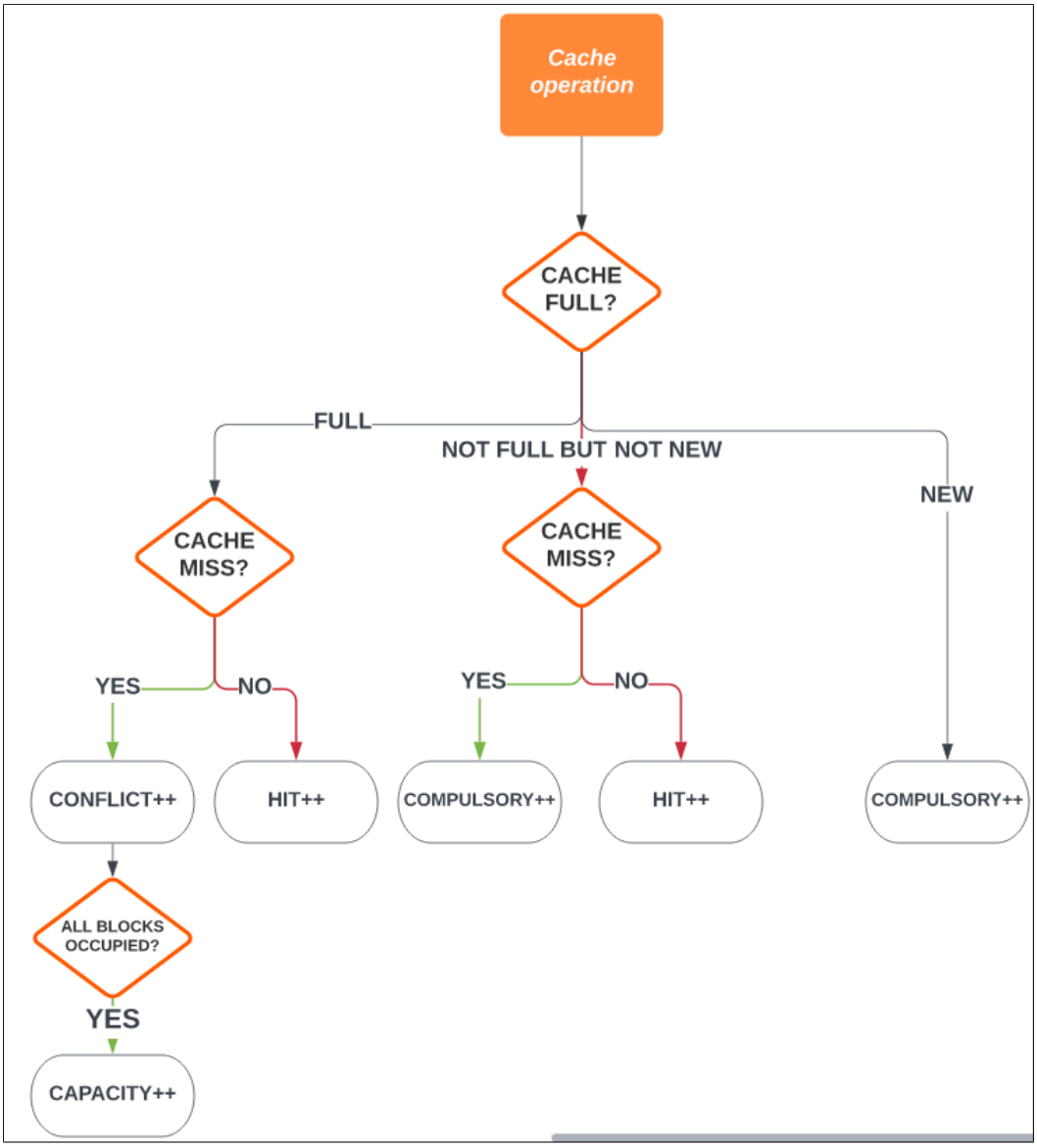


Figure 6: *Computing hits and misses(Conflict, Capacity and Compulsory)*

5.5 Challenges

1. Initially, we faced challenges in deciding the structure of the cache. It was challenging to decide how the cache could be constructed such that it is dynamically modifiable. We solved this problem by using complex data structures of C++ such as struct, list, and vector. We divided the memory address into tag bits, index bits, offset bits, and memory address using struct. Associative blocks are implemented using a list in C++, and for the whole cache, we used a vector of the above lists.

2. The solution to the above problem also solved another challenge we faced in dynamically constructing the cache using the associativity given in the problem. In this implementation, we limited the construction of elements in a set more than associativity.

3. While devising the code structure for performing the replacement for LRU, we faced a challenge in how we can track the occurrence of the address in the cache, whether least recently used or most recently used. We overcame this problem by maintaining the replacement working similar to the working of a queue in which the first element is the least recently used, and the last element is the most recent one. Whenever a conflict occurs in which the cache is full, we remove the first element and add the recent element at the end of the set.

4. We faced difficulty in keeping track of the capacity misses. To tackle this challenge, we traversed the whole cache and recorded a check flag if any of the sets have a number of blocks lesser than associativity. Then, we recorded that miss as a capacity miss.

6 Testing and Experiments

How to Run the programs To run the programme all the 4 files Cache.cpp, Cache.h and Main.cpp should be in one directory. To compile these file run the below command in that directory terminal.

```
g++ -std=c++11 Cache.cpp main.cpp  -o hw
```

Now we have an executable file hw. To run hw we need to give trace file name, capacity in bytes, block size in bytes, associativity, replacement policy input data as command line arguments in the terminal. For replacement policy if you want to test LRU input 0, for FIFO input 1, and for LIFO input 2.

```
./hw example.txt 2048 64 4 2
```

Testing We will test the code on the trace file which is actually generated by program executing and for checking that our code is correct we will run the same file on both simulators so that we can verify.

Level	Requests	Hits	Hit Rate	Average Access Time
L1	150	132	88.00%	10ns
Main Memory	18	18	100.00%	100ns
Overall	**150**	**132**	**88.00%**	**22.00ns**

The trace file for this is uploaded with the code its name is trace2.txt this picture is from online simulator **LINK** the same file is also tested for our simulator and result are the same. Here configuration was 2048 cache size, 64 block size, 4 way associative, LRU replacement policy. We didn't find the simulator for checking capacity, conflict, or compulsory misses. And also for other replacement policies. But by definition and correctness of the code we can use it for analysis.

```
NUM ACCESSES: 150

HIT RATE: 0.88
-----HIT-----
READ            88
WRITE           44
TOTAL           132

-----MISSES-----
                COMPULSORY      CONFLICT        CAPACITY        TOTAL
READ            9               0               0               9
WRITE           9               0               0               9
TOTAL           18              0               0               18
```

We have seen how to run the simulator but for checking we will need a different format trace file. It is already generated and saved as trace.txt you can open it and copy paste the content into the website after choosing cache parameters.

The Experiment

For FIFO replacement policy we will generate data for different block size, capacity and associativity.

For associativity=1, Block size=64 bytes, capacity=2048 bytes

```
HIT RATE: 0.87446
-----HIT-----
READ            25603
WRITE           12422
TOTAL           38025

-----MISSES-----
                COMPULSORY      CONFLICT        CAPACITY        TOTAL
READ            21              4247            4178            4268
WRITE           11              1180            1151            1191
TOTAL           32              5427            5329            5459
```

For associativity=2, Block size=64 bytes, capacity=2048 bytes

```
HIT RATE: 0.898721
-----HIT-----
READ            26373
WRITE           12707
TOTAL           39080

-----MISSES-----
                COMPULSORY      CONFLICT        CAPACITY        TOTAL
READ            18              3480            3459            3498
WRITE           14              892             881             906
TOTAL           32              4372            4340            4404
```

For associativity=4, Block size=64 bytes, capacity=2048 bytes

```
HIT RATE: 0.897916
-----HIT-----
READ            26235
WRITE           12810
TOTAL           39045

-----MISSES-----
                COMPULSORY      CONFLICT        CAPACITY        TOTAL
READ            18              3618            3597            3636
WRITE           14              789             784             803
TOTAL           32              4407            4381            4439
```

For associativity=8, Block size=64 bytes, capacity=2048 bytes

```
HIT RATE: 0.894766
-----HIT-----
READ            26159
WRITE           12749
TOTAL           38908

-----MISSES-----
                COMPULSORY      CONFLICT        CAPACITY        TOTAL
READ            21              3691            3689            3712
WRITE           11              853             851             864
TOTAL           32              4544            4540            4576
```

For associativity=4, Block size=128 bytes, capacity=2048 bytes

```
HIT RATE: 0.893409
-----HIT-----
READ            26045
WRITE           12804
TOTAL           38849

-----MISSES-----
                COMPULSORY      CONFLICT        CAPACITY        TOTAL
READ            8               3818            3810            3826
WRITE           8               801             796             809
TOTAL           16              4619            4606            4635
```

For associativity=4, Block size=256 bytes, capacity=2048 bytes

```
NUM ACCESSES: 43484

HIT RATE: 0.887154
-----HIT-----
READ            25696
WRITE           12881
TOTAL           38577

-----MISSES-----
                COMPULSORY      CONFLICT        CAPACITY        TOTAL
READ            4               4171            4168            4175
WRITE           4               728             728             732
TOTAL           8               4899            4896            4907
```

For associativity=4, Block size=64 bytes, capacity=1024 bytes

```
HIT RATE: 0.851716
-----HIT-----
READ            24769
WRITE           12267
TOTAL           37036

-----MISSES-----
                COMPULSORY      CONFLICT        CAPACITY        TOTAL
READ            7               5095            5091            5102
WRITE           9               1337            1336            1346
TOTAL           16              6432            6427            6448
```

For associativity=4, Block size=64 bytes, capacity=512 bytes

```
HIT RATE: 0.797443
-----HIT-----
READ            23210
WRITE           11466
TOTAL           34676

-----MISSES-----
                COMPULSORY      CONFLICT        CAPACITY        TOTAL
READ            3               6658            6658            6661
WRITE           5               2142            2140            2147
TOTAL           8               8800            8798            8808
```

Least Recently Used(LRU)

For associativity=4, Block size=64 bytes, capacity=2048 bytes

```
HIT RATE: 0.910726
-----HIT-----
READ            26579
WRITE           13023
TOTAL           39602

-----MISSES-----
                COMPULSORY      CONFLICT        CAPACITY        TOTAL
READ            18              3274            3260            3292
WRITE           14              576             572             590
TOTAL           32              3850            3832            3882
```

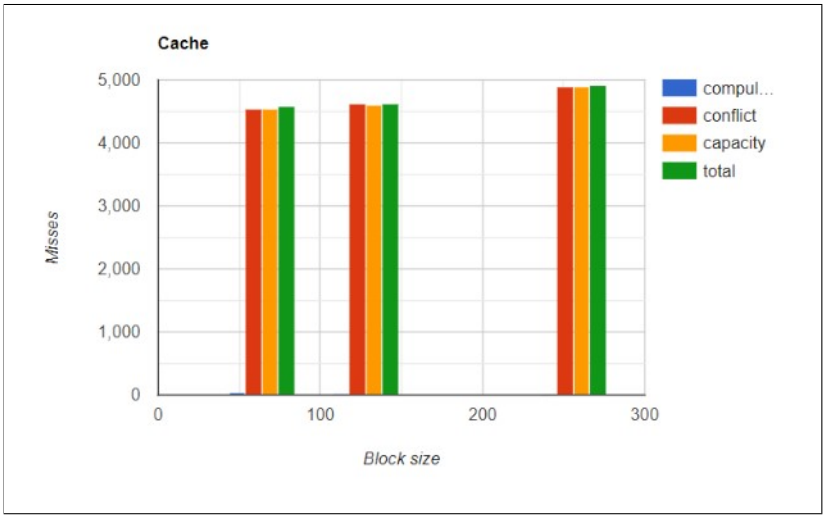
Last In First out (LIFO)

For associativity=4, Block size=64 bytes, capacity=2048 bytes

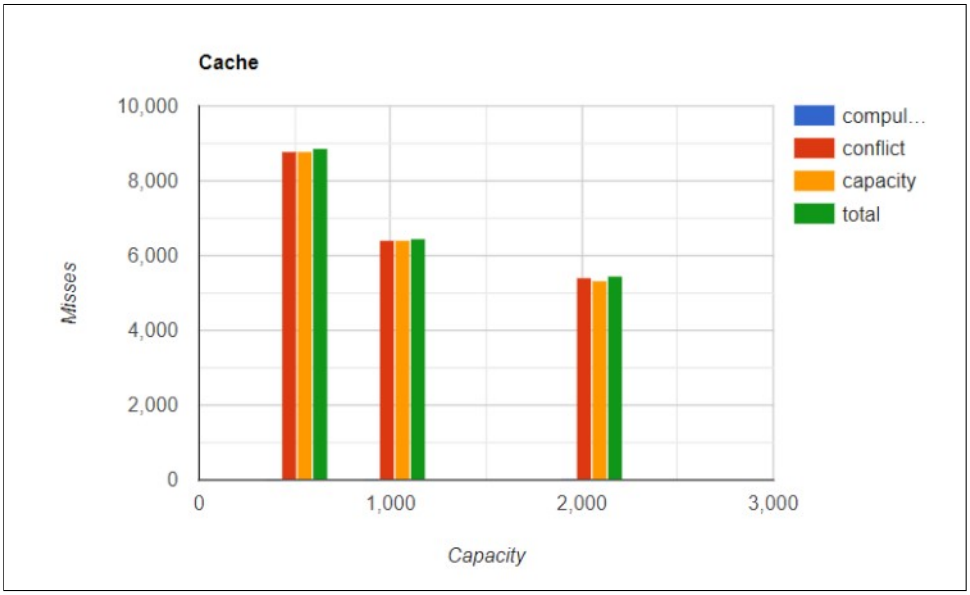
```
HIT RATE: 0.791946
-----HIT-----
READ            22802
WRITE           11635
TOTAL           34437

-----MISSES-----
                COMPULSORY      CONFLICT        CAPACITY        TOTAL
READ            18              7051            7037            7069
WRITE           14              1964            1960            1978
TOTAL           32              9015            8997            9047
```

7 Data Analysis

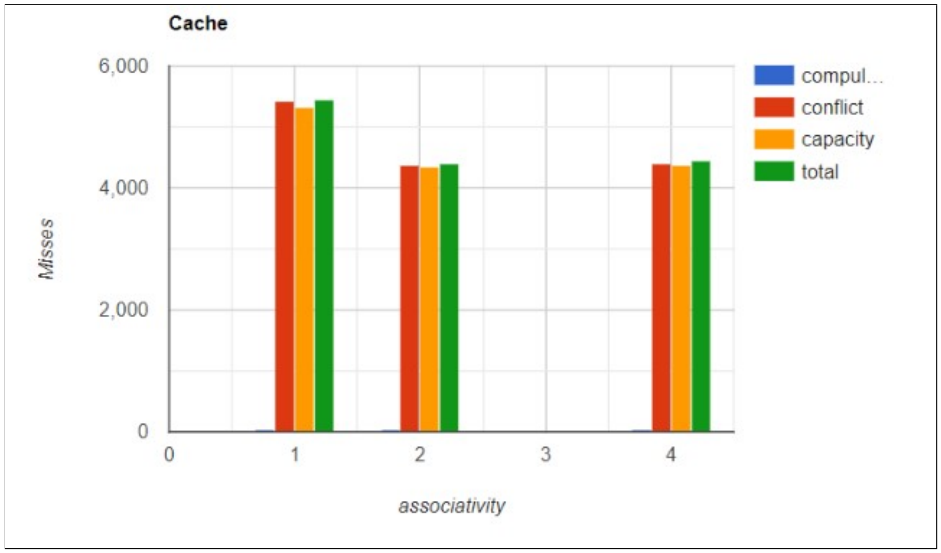
When associativity was constant, capacity was constant, replacement policy was constant and we are varying block size.



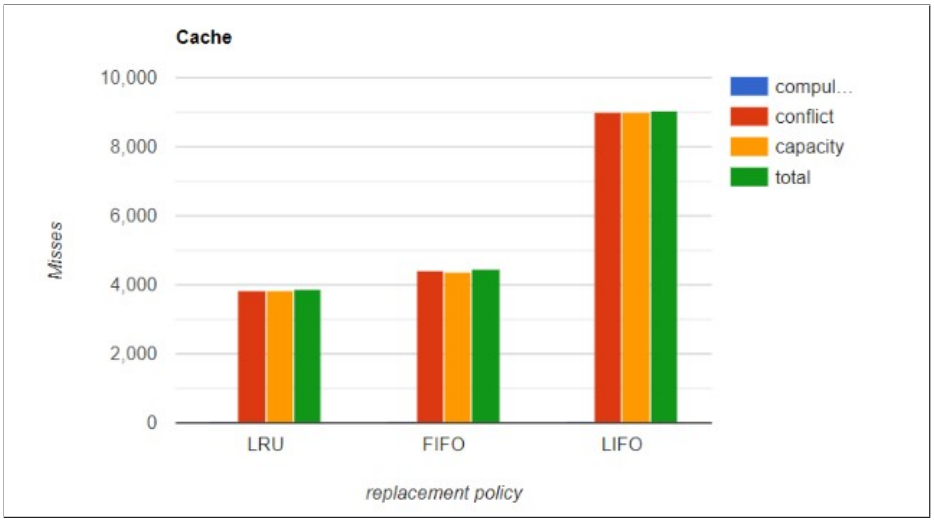
When associativity was constant, block size was constant, replacement policy was constant and we are varying capacity.



When capacity was constant, block size was constant, replacement policy was constant and we are varying associativity.



When we are changing replacement policy and all other things are constant.



OBSERVATIONS:

1. When we are increasing block size misses are slightly increasing. It is because the number of blocks are decreasing when we lower the number of blocks cache misses will be higher.

2. When we are increasing the capacity misses are reducing because the number of blocks are increasing. But it doesn't mean that cache performance will increase. There is a tradeoff between cache size and its access time so if we increase capacity then its access time will also increase.

3. When we vary the associativity in our case, misses are slightly decreasing but it can increase after some point. There is no direct relationship between associativity and total misses but there is a relationship between associativity and conflict misses if conflict misses are too high than capacity misses then increasing associativity will decrease total misses and vice versa.

4. In the fourth graph we can see that LRU for our program is much better than FIFO and LIFO. This is because of computer program access patterns.

8 Conclusion

Project does not give exact hardware implementation but it works like exact hardware. There are many configurations of the cache by varying its parameters. We got to know more about its functionality and optimization of the cache. We know about the evolution of cache and its complexity increases year by year. Optimization of the cache is as necessary as optimization of the processor. Performance of the CPU is mostly dependent on both these aspects. The project code is written such that in future we can easily add up new features easily. We have not implemented a multi level cache but it can be easily done using this project. Here we have not calculated time to access it can be done by giving penalty time and response time. By completing the project we got to know the cache memory hierarchy better than before. We got hands-on experience for implementing the various replacement policies like LRU, LIFO and FIFO and the read and write back policy.

9 References

- Maurice Wilkes introduced Memory caching
- Cache Wikipedia
- YouTube Video Link
- LRU Cache simulator

GITHUB LINK FOR THE PROJECT