Lacatus Arpad-Alex
Group 30433

Simulating the functionality of a memory cache

Contents

Lacatus Arpad-Alex
Group 30433

1.Introduction

1.1 Context

The goal of this project is to design, implement and test a simulation of a cache memory with its operations. The operations to be simulated are the writing, reading, and updating. For the updating operation we will look into specific algorithms in order to get a better grasp of the functionality of the cache. This simulation can be used to better understand how a cache memory works and to simulate one's cache memory beforehand. The visualization of the operations would also provide a simpler way of working with the cache memory, thus constituting a more efficient way to implement application that utilize the cache memory.

1.2 Specifications

The app will be implemented in Java. It will be able to represent the memory through a table, the user can choose the which instruction to be written or searched in the cache. The user has the possibility to choose the dimensions and specifications of the cache memory. The app will get the instruction from the user and by a search organized in tag index and data it will be decided if it is a miss or a hit. The number of hits and misses will be saved, and the cache updated so the process could be redone.

1.3 Objectives

Design and implement a way to simulate a cache memory by highlighting its operations. The goal of this app is to give the user a visualization of the cache memory and provide a reliable, complex, yet easy to grasp simulation of the cache.

2.Bibliographic study

Cache memory is a chip-based computer component that makes retrieving data from the computer's memory more efficient at the cost of being significantly smaller.

Data in primary memory can be accessed faster than secondary memory but still, access times of primary memory are generally in few microseconds, whereas CPU can perform operations in nanoseconds. Due to the time lag between accessing data and acting of data performance of the system decreases as the CPU is not utilized properly, it may remain idle for some time. To minimize this time gap new segment of memory is Introduced known as Cache Memory. [1]

When an operation is done on the cache two things can happen while accessing it:

- Cache hit: data is found in the cache

- Cache miss: data is not found

Types of cache memory

- **Level 1 Cache:** It is the first level of cache memory that is located inside the processor. It is found in a small amount inside every core of the processor separately. The size of this memory ranges from 2KB to 64 KB.
- **Level 2 Cache:** It is the second level of cache memory that may present inside or outside the CPU. It may not be present inside the core; in this case it can be shared between two cores depending on the architecture and is connected to a processor with the high-speed bus. The size of memory ranges from 256 KB to 512 KB.
- **Level 3 Cache:** It is the third level of cache memory that is present outside the CPU and is shared by all the cores of the CPU. May be found in some high processors. This cache is used to increase the performance of the L2 and L1 cache. The size of this memory ranges from 1 MB to 8MB.[2]

Updating Algorithms

Least Frequently Used (LFU): This cache algorithm uses a counter to keep track of how often an entry is accessed. The entry with the lowest count is removed first. This method isn't used that often, as for items that have high access time that wasn’t accessed recently is not considered.

Least Recently Used (LRU): This cache algorithm keeps recently used items near the top of cache. When the new item is accessed, it is placed at the top of the cache. When cache reaches its capacity limit, the less recently accessed items will be removed starting from the bottom of the cache. This is a demanding algorithm for the memory, as it stores the time when an item was accessed last. In addition, when there is a deletion all last access times need to be updated.

Adaptive Replacement Cache (ARC): Developed at the IBM Almaden Research Center, this cache algorithm considers both LFU and LRU, as well as evicted cache entries to get the best use out of the available cache.

Most Recently Used (MRU): This cache algorithm removes the most recently used items first. A MRU algorithm is suitable for situations in which we predict that an older item has a higher chance of being accessed.[2]

Types of organizing the cache

Direct Mapping: Maps each block of main memory into only one possible cache line. or In Direct mapping, assign each memory block to a specific line in the cache. When there is a collision, the old information is replaced by the new one. An address space is split into two parts index field and a tag field. The cache is used to store the tag field whereas the rest is stored in the main memory. Direct mapping`s performance is directly proportional to the Hit ratio.
Associative Mapping: The associative memory is used to store content and addresses of the memory word. Any block can go into any line of the cache. The id bits are used to identify which word in the block is needed, the remaining bits are the tag. By doing this we can place any information anywhere in the cache. It is the fastest and the most flexible mapping form.

Set-Associative Mapping: Enhanced form of direct mapping. Instead of having exactly one line that a block can map to in the cache, we will group a few lines together creating a "set". Then a block in memory can map to any one of the lines of a specific set. Each word that is present in the cache can have two or more words in the main memory for the same index address. [3]

Organization

For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory. In most contemporary machines, the address is at the byte level. The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of s-r bits (most significant portion) and a line field of r bits. This latter field identifies one of the $m=2^r$ lines of the cache.[4]

3.Analysis

3.1 Project proposal

Working with cache memory can be sometimes confusing and hard to pick up at the beginning due to the complex way in which is implemented, but one needs to fully understand it to get to the optimal solution to a certain problem.

The goal of this project is to provide an easy to grasp simulation of the cache memory and a way to test the different kinds of cache by changing the update algorithm and dimensions of the

cache. The simulations that are run with this project should provide an efficient and reliable way of visualizing the cache.

The result of the project will be an app with a graphical interface that by processing the information provided by the user will simulate a cache memory and present the results on the screen. The results will be the number of cache hits and misses up until then and if the current instruction was a hit or a miss. The app can be simulated with any updating algorithm chose by the user. The method of mapping implemented will be directly mapped.

3.2 Project plan

Phase one:

- Weeks 2-3: Introduction, Bibliographic Study
- Weeks 4-5: Project proposal, Project Plan, Problem Analysis

Phase two:

- Weeks 6-9: Design + Implementation
- Weeks 10-11: Testing

Phase three:

- Weeks 12-13: Final touches + finishing documentation

3.3 Use case diagram + flow chart

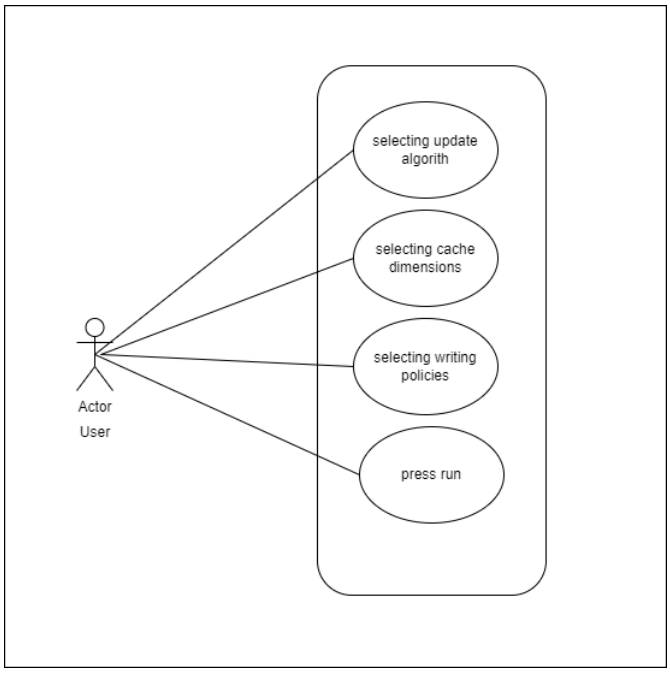


fig 1 Use case diagram

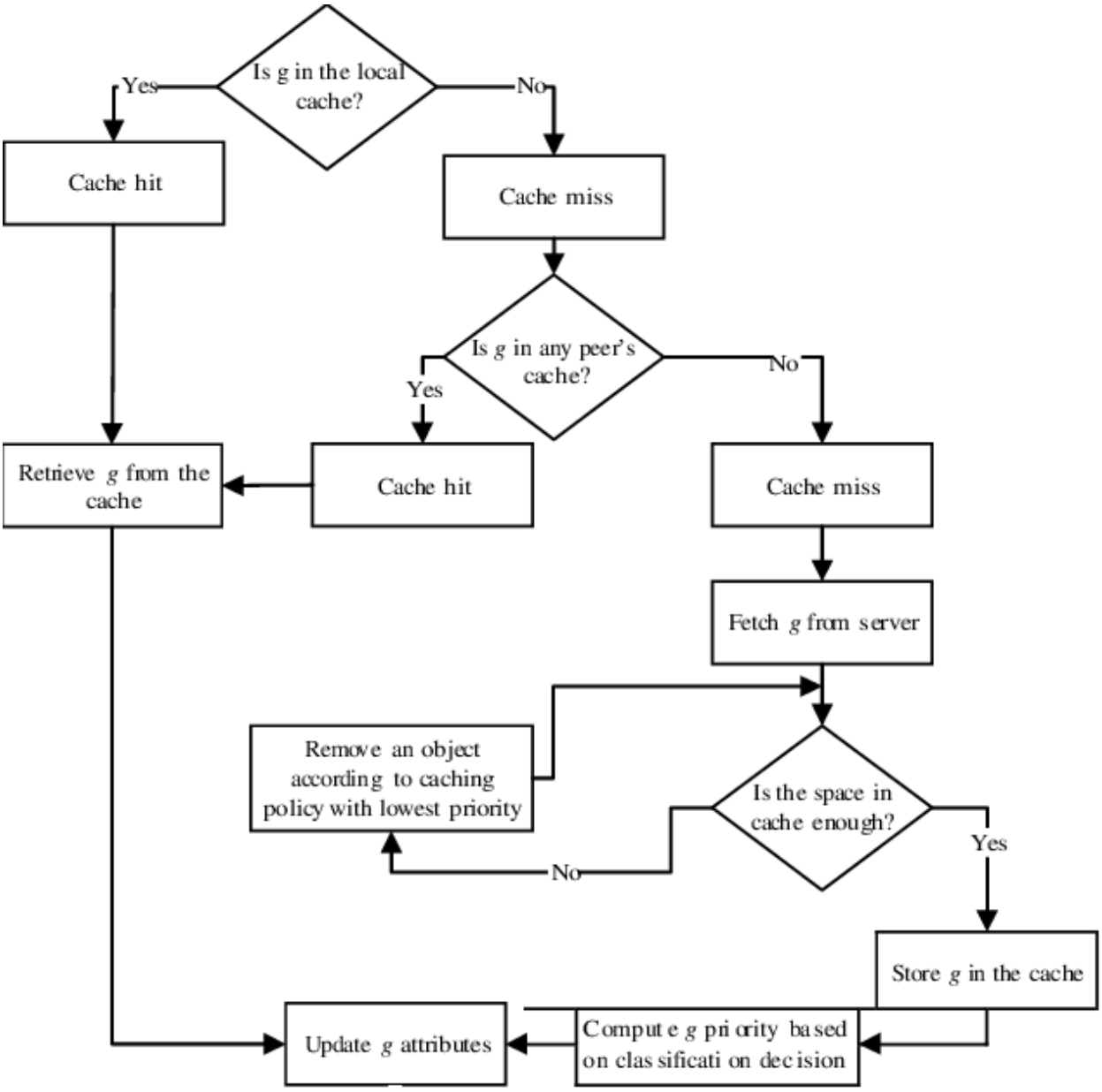


Fig2 cache algorithm flowchart [5]

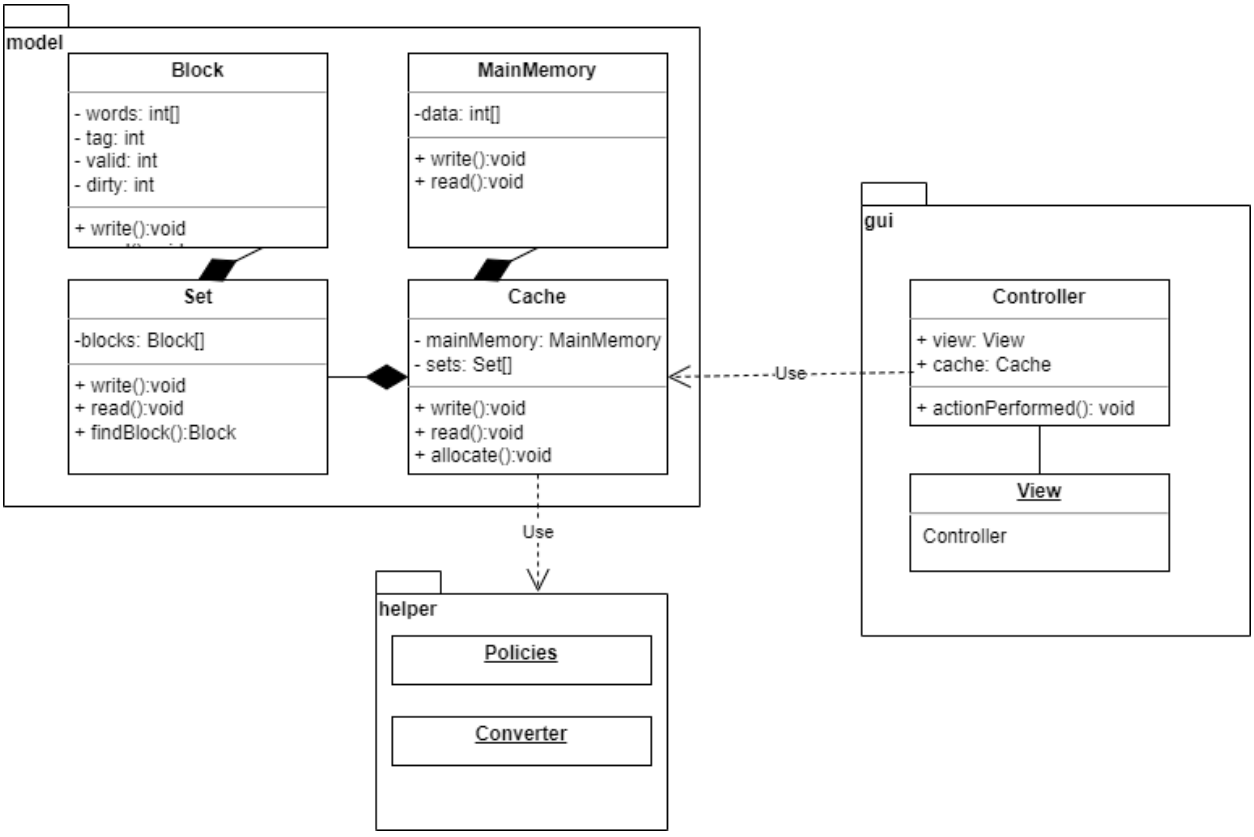
4.Design

The main parts of the cache simulator are the cache and the gui elements.

The cache class consists of the main memory and an array of set that form the cache. This class is the key part of our simulation as all the operations are done on it or on the components. The main operations here are read and write that try to modify the cache with the help of the function

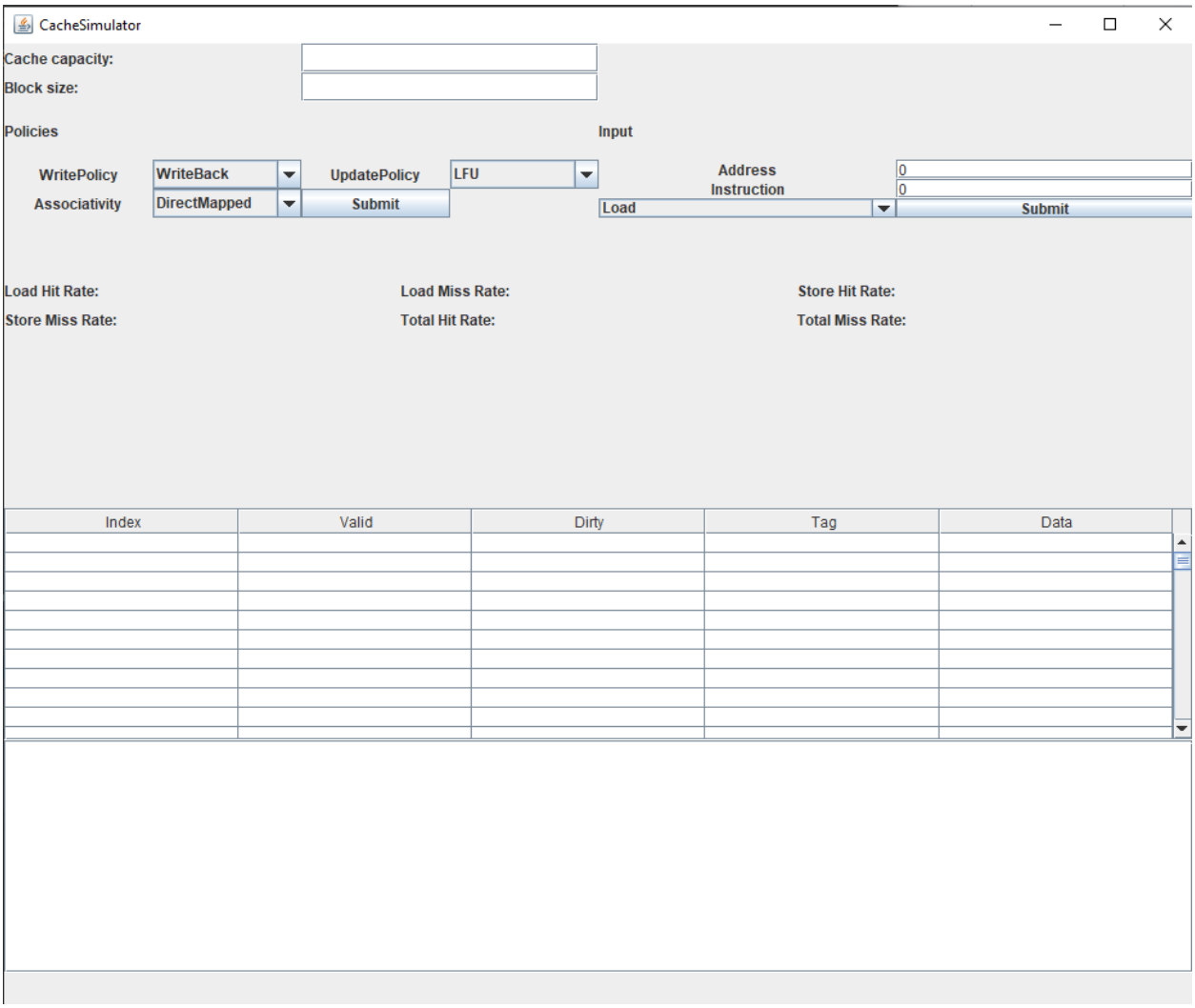
allocate. This function has the purpose of evicting a chosen block to the main memory, the way of choosing the evicted is done with a update policy and the way in which we write the main memory is decided by a write policy. The set class has functions that provide us the blocks desired for evicting requiring an update policy, moreover has a findBlock function that takes a tag as argument and read and write functions that operate on blocks addressed via tag and offset.The set has an array of blocks. The block class provides fields such as tag, dirty, valid, words that can be accessed for read or write purposes(words).The main memory simulates a memory different from the cache where the data is written back through evictions. It can be visualize through print and read functions and updated through write.

For the GUI part we have a model-view controller. The view takes care of the visual part of our application while the controller manages the callbacks and provides the connection with our models. It provides the user a way of choosing parameters regarding our cache and provides the result to the view.



Here we have the class diagram and we can see the dependencies between classes as well as the packages. Model package has the class that model our application, gui the classes the classes that make the graphical user interface and the helper has utility classes.

Lacatus Arpad-Alex
Group 30433



This is how the gui looks like for our app. It has fields that help the user to format the cache(capacity , block size, associativity), to choose policies (write, update) and provide a way to insert data into our cache.

The user also can see the number of hits/misses for each operation , the cache (represented in a table) and the main memory (represented in a textbox).

7.Implementation

The implementation was done with the goal of writing modular, easy to understand code so we have the following implementations:

Lacatus Arpad-Alex
Group 30433

```
1 usage
public void write(int address, int data, UpdatePolicy updatePolicy,WritePolicy writePolicy) {

    // if the block isn't found in the cache, put it there
    if ( !isInMemory(address) ) {
        allocate(address, updatePolicy,writePolicy);
        numWriteMisses++;
        if(writePolicy.equals(WritePolicy.WRITE_AROUND))
            return;
    }

    // calculate what set the block is in and ask it for the data
    int index = ( address / blockSize ) % sets.length;
    int blockOffset = address % blockSize;
    Set set = sets[index];

    numWrites++;
    set.write( getTag(address), blockOffset, data );
}
```

Here we have the implementation for the write function from the cache class. If the block is not in memory yest we allocate it with the right policies and update the misses number. After that w write in our cache the data. To note that for the write around policies we skip this step if the block is not found. The read function works similarly.

```
private void allocate(int address, UpdatePolicy updatePolicy, WritePolicy writePolicy) {

    int index = ( address / blockSize ) % sets.length;
    Set set = sets[index];
    Block blockToBeEvicted=null;
    switch (updatePolicy){
        case LRU ->blockToBeEvicted = set.getLRU();
        case LFU ->blockToBeEvicted = set.getLFU();
        case ARC ->blockToBeEvicted = set.getBestForARC();
        case MRU ->blockToBeEvicted = set.getMRU();
    }


    switch(writePolicy){
        case WRITE_BACK -> writeBack(address,blockToBeEvicted,index);
        case WRITE_AROUND -> writeAround(address,blockToBeEvicted,index);
        case WRITE_THROUGH ->writeThrough(address,blockToBeEvicted,index);

    }

}
```

Looking at the implementation of the allocate function from the cache class we can see that by providing the right policies we get the block to be evicted from the set and we write to the main memory.

```
1 usage
public Block getLRU() {
    int index=0;
    int least=blocks[index].getRecentUse();
    for (int i = 1; i < associativity; i++) {
        if(least>blocks[i].getRecentUse())
        {
            index=i;
            least=blocks[i].getRecentUse();
        }
    }
    Block blockToReturn = blocks[index];
    return blockToReturn;
}
```

Taking a peek at the implementation of getLRU(get least recentlu used) we can notice the way in which the set searches for the evictee and returns it. The other functions that work with different policies work similarly.

7.Testing and validation

For the testing there were provided values for the cache in the gui. We input a wide range of loads/stores in our cache and we came to the conclusion that it works as intended. To be noted that for testing we looked at:

-how tags interact in our cache

-if valid/dirty bits work as intended

-if main memory is updated when needed

-the accuracy of the mis/hit stats provided

-how the app behaves with different policies

7.Conclusions

The goal of this project was to create an app that can simulate a cache memory and provide its users an easy to grasp visualization of the cache. This app gives the user a way of interacting with the different kinds of cache implementations and algorithms that are used in this memory.

I enjoyed working on this project due to the fact that completing it involved working and understanding a complex, yet interesting concept. I consider that this experience helped me get more accustomed with how the memory works in computer systems and will for sure prove itself useful in the projects to come.

8.Bibliography

[1] Linsey Knerl - https://www.hp.com/us-en/shop/tech-takes/what-is-cache-memory

[2] Harsh Shukla - https://www.geeksforgeeks.org/cache-memory/

[3] Ben Lutkevich - https://www.techtarget.com/searchstorage/definition/cache-memory

[4] Kim Hefner - https://www.techtarget.com/searchstorage/definition/cache-algorithm

[5] Waheed Yasin - https://www.researchgate.net/figure/A-general-flow-chart-of-the-proposed-intelligent-cooperative-web-proxy-caching-approach_fig2_274640931