Structure of Computer Systems
Cache Memory Simulation

Student: Elekes Péter

University: Technical University of Cluj-Napoca

Faculty: Computer Science

Year: III.

Group: 30432

Table of Contents

Introduction

The Cache Memory is used by the CPU to speed up the process of acquiring data from the memory. It is situated between the CPU and the memory. There are multiple cache levels, such as L1, L2 and so on. It is mainly using SRAM that is way quicker than DRAM, but therefore it is more expensive. This is the reason why SRAM memories are smaller in size. The main purpose of the cache memory is that frequently accessed data is copied in the cache, thus reducing the time needed to access the information.

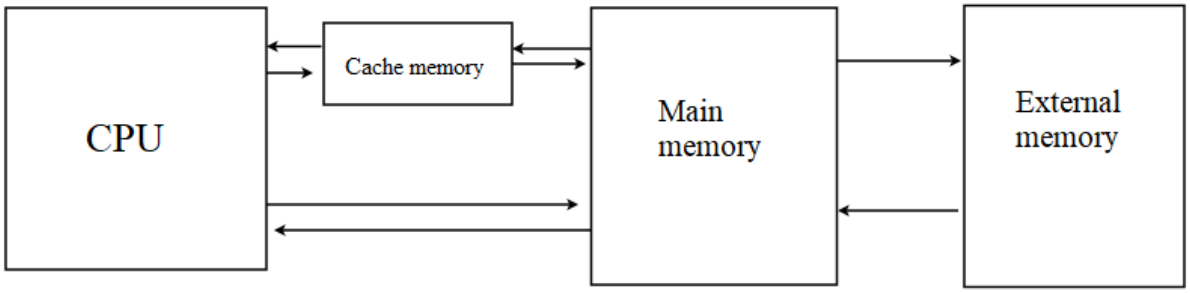

Figure 1 – visualizing the architecture [4]

Bibliographic research

If the CPU wants to read or write to the RAM it first checks whether the data is copied to the cache. If it happens to be there than the CPU uses the cache instead, because it is way faster. There are three independent caches that are used often today:

1. Instruction cache – to speed up the instruction fetch
2. Data cache – to speed up data fetch/store (organized in more levels e.g.: L1, L2 …)
3. Translation Lookaside Buffer – to speed up virtual-to-physical address translation

If the processor finds the memory location in the cache we call it a cache hit, but if the CPU does not find the memory location we call that a cache miss. If a miss happens, the cache allocates a new entry and copies in data from main memory. This can be measured by what we call a hit ratio.

$$hit\ ratio = \frac{(nr.of\ hits)}{nr.of\ hits + misses}$$

We can compute the miss ratio similarly, if we subtract from one

$$miss\ ratio = 1 - hit\ ratio$$

Cache structure

The cache row entries have this structure:

tag	data block	flag bits

The data block contains the actual information, the tag contains the address of the data. The flag bit is a so-called ‘valid’ bit. The valid bit indicates whether or not a cache block has been loaded with valid data. The "size" of the cache is the amount of main memory data it can hold. This size can be calculated as the number of bytes stored in each data block times the number of blocks stored in the cache.

An effective memory address which goes along with the cache line (memory block) is split (MSB to LSB) into the tag, the index and the block offset.

tag	index	block offset

The index describes which cache set that the data has been put in. The index length is $\log_2(s)$ bits for s cache sets. The block offset specifies the desired data within the stored data block within the cache row. The tag contains the most significant bits of the address, which are checked against all rows in the current set to see if this set contains the requested address. If it does, a cache hit occurs. (tag_length = address_length - index_length - block_offset_length)

Example:

The original Pentium 4 processor had a four-way set associative L1 data cache of 8 KiB in size, with 64-byte cache blocks. Hence, there are 8 KiB / 64 = 128 cache blocks. The number of sets is equal to the number of cache blocks divided by the number of ways of associativity, what leads to 128 / 4 = 32 sets, and hence 25 = 32 different indices. There are 26 = 64 possible offsets. Since the CPU address is 32 bits wide, this implies 32 - 5 - 6 = 21 bits for the tag field.

The original Pentium 4 processor also had an eight-way set associative L2 integrated cache 256 KiB in size, with 128-byte cache blocks. This implies 32 - 8 - 7 = 17 bits for the tag field.

Cache Placement Policies

There are multiple placement policies that decide where a particular entry of the main memory can go. The most basic way to organize the cache memory can be done by using Direct-Mapped Cache where each location in the main memory can go in only one entry in the cache. We can also call this approach “one-way set associative” because of this reason. Direct-mapped cache has a good best-case time but it is unpredictable in the worst case. The performance of direct

mapping is directly proportional to the hit ratio. On the contrary, if we can put any data anywhere, we call that approach fully associative. Choosing the right value of associativity involves a trade-off. If there are multiple places a piece of data can be stored, the CPU has to check in every such place. The general guideline is that doubling the associativity, from direct mapped to two-way, or from two-way to four-way, has about the same effect on raising the hit rate as doubling the cache size. Two-way set associative cache does the exact thing its name suggests. Each location in the main memory can be cached in either of two locations in the cache.

Cache Write Policies

A cache's write policy is the behavior of a cache while performing a write operation. It has a central part in all the variety of different characteristics exposed by the cache. I will be looking at two policies, but there are more approaches:

- write-through
- write-back

In a write-through cache, every write to the cache causes a write to main memory.

A write-back policy differs in such way, that writes are not present instantaneously on the main memory, and the cache only tracks the locations that were written over (calling them dirty).

Replacement Policies

Also referred to as cache algorithms are used to determine how data is being handled on the cache. As we can imagine, the cache is faster when more frequently used data, recent data, etc.. are on it. If the cache memory is ever filled these algorithms help choose which segment will 4 be written over.

The average memory access time is:

$$T = m \times T_m + T_h + E$$

Figure 2 - Average memory reference time

where

m = miss ratio

T_m = time to make a main memory access when there is a miss

T_h = the latency: the time to access the cache

E = various secondary effects

The "hit ratio" of a cache describes how often a searched-for item is actually found in the cache.

The "latency" of a cache describes how long after requesting a desired item the cache can return that item (when there is a hit).

Each cache algorithm strategy is a compromise between hit rate and latency. Some notable replacement policies are:

- First in first out (FIFO)
- Last in first out (LIFO)
- Least recently used (LRU)
- Most recently used (MRU)
- Least-frequently used (LFU)

In my project I will try to implement at least two o, namely FIFO and LRU. If I will have the time I also want to implement LFU.

The FIFO algorithm is self-explanatory. The cache evicts the blocks in the order they were added, without any regard to how often or how many times they were accessed before.

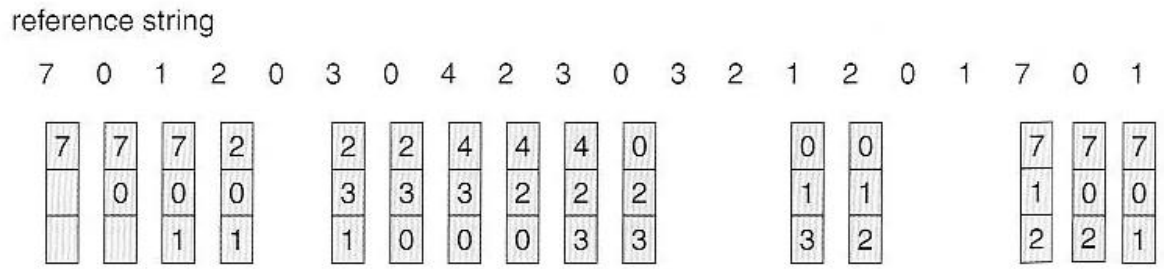

Figure 3 - FIFO algorithm [2]

The LRU algorithm discards the least recently used items first. In order for this algorithm to work efficiently, it has to keep track of time, which is expensive.

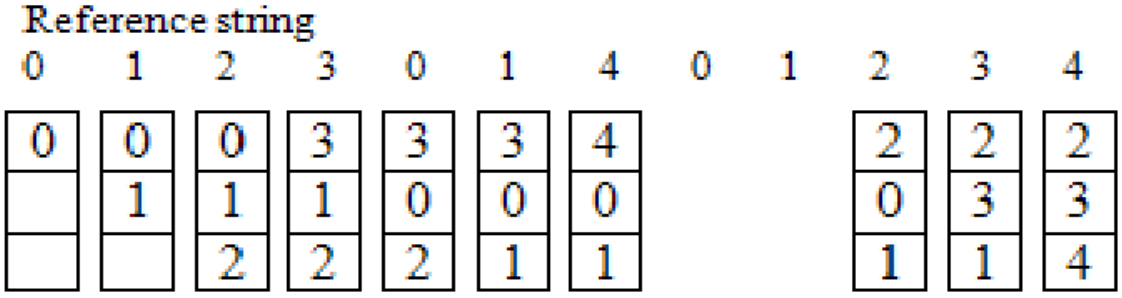

Figure 4 - LRU algorithm [2]

The LFU algorithm counts how often an item is needed. The ones requested for the least are discarded first. This works very similar to LRU except that instead of storing the value of how recently a block was accessed, we store the value of how many times it was accessed.

Project proposal and analysis

In this project I want to realize a fully functioning cache memory simulator with a Graphical User Interface. I hope to learn new information about the structure of the cache while implementing it in a high-level language.

Language

I will implement the simulator in Java, because it has many libraries that I will be able to use throughout the project. The Object-Oriented design will come in handy, because I want to implement multiple cache write policies, placement policies and replacement policies. I will approach designing the User Interface in Java Swing, because it is simple to use and yields a functional result. I'd like to think that I know Java really well, so it will ease my workflow. Having all these different components will be really hard to manage, but breaking them down to classes I think it will be manageable.

Goals

In the end I want to have a fully functional cache memory simulator with modifiable parameters. For the placement policy I will implement a n-way set associative cache, with a modifiable n. This way, the user can choose between direct-mapped cache, fully associative cache, or an n-way set

associative cache. I probably will achieve this interaction using an abstract class, or an interface in Java. In my opinion this will take the most time to get right, because I have to account for all the special cases.

Regarding the cache write policies, firstly I want to implement just one of them, namely write-through. If I have the time, and everything else works, I will try to implement the write-back policy, but I'm not promising anything.

As for the replacement policies, I want to implement a queue-based policy in the beginning, like FIFO. To use this algorithm I will probably use the List interface in Java because it can be easily manipulated to work like a FIFO. Then, I will go on to the recency-based policy LRU. For this functionality I need an additional variable that stores the time an element was last accessed so I can decide what to do with it. After I finish both I will implement a frequency-based policy too, my choice being LFU. This algorithm is quite similar to the LRU, as I stated before. I will have a counter as an additional variable that counts how many times a piece of data was accessed.

Use Case

The program will be interactive in traceable step-by-step.

1. The user will be able to input the size of the cache, the block size and memory size
2. The user will choose write, place and replace policies
3. The user will input the data they want to read/write, and press the corresponding button
4. The user will be able to trace the program with the history and hit/miss counter

Realization

With these goals in mind, I think I will start by implementing all of the approaches on their own. I want to provide the user with an interactive program that can be traced step-by-step. Memory addresses will be randomly generated. If I want to find a value I can just instantly check for it using built-in methods. Java and its libraries can be powerful for these tasks. An instruction history will also be present so the user can trace all the steps taken. Graphical representations of both the cache and the main memory will be visible so the user can see where a value was written/read from. I hope I can design an intuitive, yet good-looking UI that is functional. Java Swing isn't famous for its beautiful design elements, but there are ways to make a program look pretty.

Design

MVC design pattern

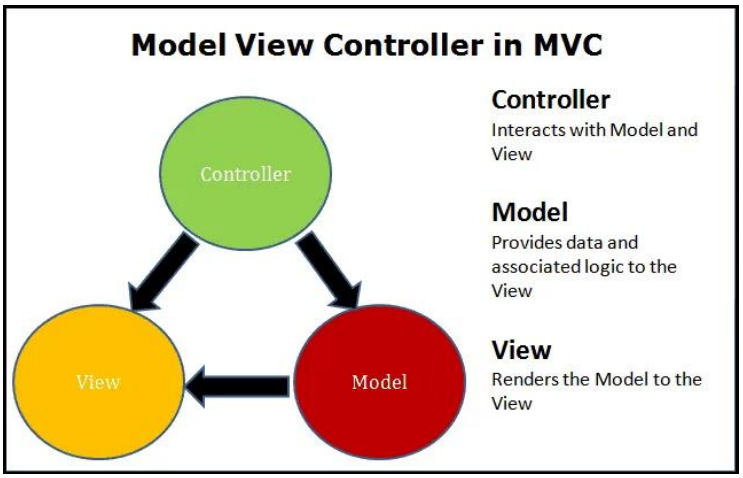

Figure 5 - The MVC Strategy

The first step I took was to create the architectural pattern. I declared the packages necessary (Model, View and Controller). I also added the Main.java file, which only calls the initialize method from the Controller class. To introduce you to the MVC architecture design I will talk a bit about it. We have to split all the components of the project into 3 main packages: Model, View and Controller. Each of these elements have a separate role they have to fulfill. The Model has to manage the data and the logic of the application, the View describes the way the Model is displayed in the UI and the Controller takes user input and converts it into a command that is then forwarded to the Model. This creates a circular motion of logic between the packages: the user interfaces with the View, which then passes the call to the Controller, which manipulates the Model, which in turn creates events to pass back to the View. This approach is really efficient for code re-use and parallel development.

Model

The Model package in this Java application contains classes and enums related to cache memory simulation. The package includes the following:

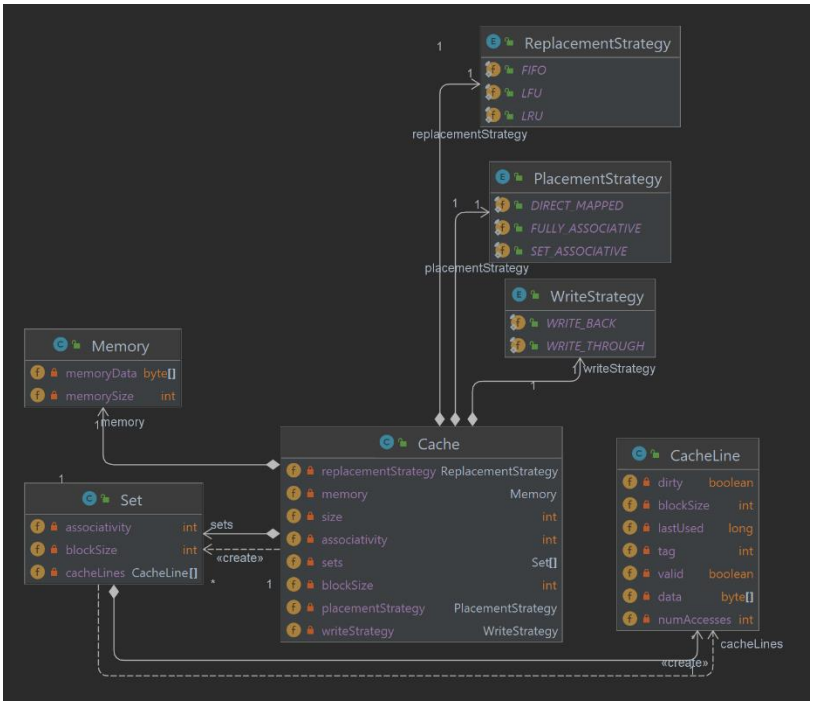

Figure 6 - UML Diagram of the Model Package

Enums:

WriteStrategy

The WriteStrategy enum represents the different strategies that can be used when writing to the cache. The available strategies are:

WRITE_THROUGH: In this strategy, the data is written to both the cache and the main memory simultaneously.

WRITE_BACK: In this strategy, the data is only written to the cache. The data is then marked as dirty, indicating that it has been modified and needs to be written to the main memory at a later time.

ReplacementStrategy

The ReplacementStrategy enum represents the different strategies that can be used when replacing a cache line in the cache. The available strategies are:

LRU (Least Recently Used): In this strategy, the cache line that has been in the cache the longest without being accessed is replaced.

LFU (Least Frequently Used): In this strategy, the cache line that has been in the cache the oldest without being accessed is replaced.

FIFO (First In First Out): In this strategy, the cache line that was added to the cache first is replaced.

PlacementStrategy

The PlacementStrategy enum represents the different strategies that can be used when placing a new cache line in the cache. The available strategies are:

DIRECT_MAPPED: In this strategy, each block of data is associated with a specific cache line.

FULLY_ASSOCIATIVE: In this strategy, any block of data can be placed in any cache line.

SET_ASSOCIATIVE: In this strategy, blocks of data are grouped into sets, and each set has a number of cache lines available for use.

Classes

Set

The Set class represents a set in the cache. A set is a group of cache lines that hold data for a specific address range.

Cache

The Cache class represents the cache memory. It consists of multiple sets and has a specified size and associativity.

CacheLine

The CacheLine class represents a cache line in the cache. A cache line is the smallest unit of data that can be stored in the cache and holds a block of data from the main memory.

Memory

The Memory class represents the main memory of the system. It holds all the data that is not present in the cache.

View

The SimulationView class represents the main window of the application. It is responsible for creating and laying out the various UI elements of the window, such as buttons, labels, and text fields. It also handles user input and updates the display as necessary.

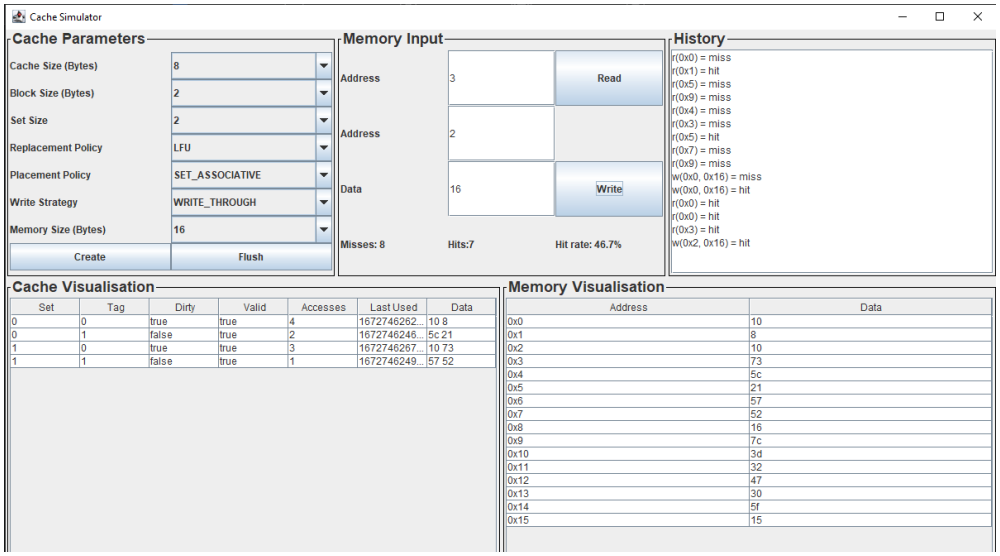

Figure 7 - Screenshot of the application

Controller

The ViewController class is the main controller of the application. It is responsible for handling user input, checking for its validity, and running the program. It communicates with the Model package to generate the cache and retrieve data as needed, and with the View package to update the display.

The ViewController class contains several methods to handle different user actions and events. For example, it contains methods to handle the input from the user. It also contains methods to validate user input, such as checking that a text field contains a valid number.

Implementation

Implementing this project was no easy task, but perhaps I will start documenting the easier things and we can make our way from there.

The SimulationView class is responsible for creating and laying out the various UI elements of the main window of the application. It uses a GroupLayout to create a well-balanced and visually appealing layout. In the SimulationView class, various UI elements such as combo boxes, text areas, and buttons are created and added to the window. Two tables are also generated to represent the cache and main memory, respectively. So far, so good.

The enums PlacementStrategy, ReplacementStrategy, WriteStrategy are all created so I can communicate between multiple classes how the cache memory should work.

Let’s start with the building blocks of this project, the cache lines. The CacheLine class represents a cache line in the cache. A cache line is the smallest unit of data that can be stored in the cache and holds a block of data from the main memory. The following fields can be found in a Cache Line:

- tag: An integer representing the tag of the cache line. The tag identifies the block of data that the cache line holds
- blockSize: An integer representing the size of the block of data that the cache line holds.
- dirty: A boolean indicating whether the data in the cache line has been modified and needs to be written back to the main memory.
- valid: A boolean indicating whether the cache line contains valid data.
- data: A byte array holding the block of data that the cache line represents.
- numAccesses: An integer representing the number of times the cache line has been accessed.
- lastUsed: A long integer representing the timestamp of the last time the cache line was accessed.

Methods:

- int read(int address): Reads a byte from the cache line at the specified address. Updates the numAccesses and lastUsed fields. Returns the byte read.
- void write(int offset, byte data): Writes a byte to the cache line at the specified offset. Sets the valid and dirty fields to true. Updates the numAccesses and lastUsed fields.
- void writeLine(int tag, int offset, byte[] data): Writes a block of data to the cache line. Sets the tag, data, valid, and dirty fields. Sets the numAccesses and lastUsed fields to 0 and the current timestamp, respectively.
- various setters and getters.

Next in the hierarchy would be the Set class. This class represents a set in the cache. A set is a group of cache lines that hold data for a specific address range.

Fields

- associativity: An integer representing the associativity of the set. The associativity determines the number of cache lines in the set.
- blockSize: An integer representing the size of the blocks of data that the cache lines in the set can hold.
- cacheLines: An array of CacheLine objects representing the cache lines in the set.

Methods

- Set(int associativity, int blockSize): Constructs a new Set object with the specified associativity and block size. Initializes the cacheLines array with new CacheLine objects.
- CacheLine findCacheLine(int tag): Searches the cacheLines array for a CacheLine with the specified tag and returns it if found. Returns null if no such CacheLine is found.
- int read(int tag, int offset): Reads a byte from the cache line with the specified tag at the specified offset. Returns the byte read.
- void write(int tag, int offset, byte data): Writes a byte to the cache line with the specified tag at the specified offset.
- CacheLine getLRU(): Returns the least recently used CacheLine in the cacheLines array.
- CacheLine getLFU(): Returns the least frequently used CacheLine in the cacheLines array.
- CacheLine getFIFO(): Returns the first CacheLine in the cacheLines array that has a tag of -1, or the first CacheLine in the array if no such CacheLine is found.
- various getters and setters

Now to the class that holds them all together, the Cache class.

Fields

- size: An integer representing the size of the cache in bytes.
- blockSize: An integer representing the size of the blocks of data that the cache can hold.
- associativity: An integer representing the associativity of the cache. The associativity determines the number of cache lines in each set.
- memory: An instance of the Memory class representing the main memory.
- placementStrategy: An enumeration value of the PlacementStrategy type representing the placement strategy of the cache. The placement strategy determines how data is mapped to cache lines.
- replacementStrategy: An enumeration value of the ReplacementStrategy type representing the replacement strategy of the cache. The replacement strategy determines which cache line to replace when the cache is full.
- writeStrategy: An enumeration value of the WriteStrategy type representing the write strategy of the cache. The write strategy determines how writes to the cache are handled.
- sets: An array of Set objects representing the sets in the cache.

Methods

- Cache(int size, int blockSize, int associativity, PlacementStrategy placementStrategy, ReplacementStrategy replacementStrategy, WriteStrategy writeStrategy, Memory memory): Constructs a new Cache object with the specified size, block size, associativity, placement strategy, replacement strategy, write strategy, and main memory. Initializes the sets array with new Set objects.
- int getTag(int address): Returns the tag of the cache line that holds the block of data at the specified address.
- void allocate(int address): Allocates a cache line for the block of data at the specified address. If all cache lines are in use, the cache line replacement strategy is used to determine which cache line to replace. If the cache line being replaced is dirty (has been modified), its data is written back to the main memory. The block of data at the specified address is then loaded into the cache line from the main memory.
- boolean read(int address): Reads a byte from the cache at the specified address. If the block of data at the address is not in the cache, it is loaded into the cache from the main memory. Returns true if the data was in the cache and false if it was not.
- boolean write(int address, byte data): Writes a byte to the cache at the specified address. If the block of data at the address is not in the cache, it is loaded into the cache from the main memory. The write strategy determines how the write is handled. Returns true if the data was in the cache and false if it was not.
- void flush(): Flushes the cache by writing back any dirty data to the main memory and invalidating all cache lines.

Lastly, The ViewController class serves as the controller in the Model-View-Controller architecture of the cache memory simulator application. It is responsible for handling user interactions, such as button clicks and form submissions, and updating the model (cache and main memory) and view (tables and history) accordingly.

When the application starts, the ViewController class initializes the view by creating the main window and all its components, such as tables, buttons, and form inputs. It then listens for user interactions and responds to them by performing the appropriate actions.

For example, when the user clicks the "Write" button, the ViewController class retrieves the input from the form, validates it, and updates the cache and main memory with the new data. It also updates the tables in the view to reflect the changes in the model.

The ViewController class also keeps track of the history of all actions performed by the user and displays it in the history table. This allows the user to see a record of all their actions and helps them debug any issues with their inputs.

Overall, the ViewController class plays a crucial role in the functioning of the cache memory simulator by tying together the model and view and handling user interactions.

Tests and validation

To check if the application runs correctly we can test each of the different ways a cache can behave.

Replacement Policies

LRU (Least Recently Used) Replacement Strategy:

Initialize a cache with 8 byte cache size, 16 byte memory size, block size of 2, set size of 2, LRU replacement strategy, set associative placement policy, write back write policy.

- Read from address 0, this should be placed in set 0
- Read from address 4, this should be placed in set 0
- Read from address 0 again, this should make it to be the most recently used block
- Read from address 8, this should replace the values stored in the cache from address 2, because they were not used so often

LFU (Least Frequently Used) Replacement Strategy:

Initialize a cache with 8 byte cache size, 16 byte memory size, block size of 2, set size of 2, LFU replacement strategy, set associative placement policy, write back write policy.

- Read from address 0, this should be placed in set 0
- Read from address 4, this should be placed in set 0
- Read from address 0 again, this should increase the number of accesses
- Read from address 8, this should replace the values stored in the cache from address 2, because they were not used so often

FIFO (First In, First Out) Replacement Strategy:

Initialize a cache with 8 byte cache size, 16 byte memory size, block size of 2, set size of 2, FIFO replacement strategy, set associative placement policy, write back write policy.

- Read from address 0, this should be placed in set 0
- Read from address 4, this should be placed in set 0
- Read from address 0 again, this should increase the number of accesses, and last time
- Read from address 8, this should replace the values stored in the cache from address 0, even though it was last accessed, but since we use FIFO it was the first element that entered the cache.

Write Policies

Write Through Write Strategy:

- Initialize a cache (it doesn’t matter what parameters)
- Try writing to a memory address
- It should write both to cache and main memory

Expected output: cache content and memory content both changed

Write Back Write Strategy:

Initialize a cache with 8 byte cache size, 16 byte memory size, block size of 2, set size of 2, FIFO replacement strategy, set associative placement policy, write back write policy.

- Write to address 0, this should be placed in the first set but value shouldn't be written to main memory
- Read from address 4, this should be placed in the first set
- Read from address 8, this should overwrite the value in the cache and write back the value written before to the main memory.

Placement Policies

Direct Mapped

Initialize a cache with 8 byte cache size, 16 byte memory size, block size of 2, set size of 2, FIFO replacement strategy, direct mapped placement policy, write back write policy.

- Read from address 0, this should be placed in set 0
- Read from address 2, this should be placed in set 1
- Read from address 4, this should be placed in set 2
- Read from address 6, this should be placed in set 3
- Read from address 8, this should be placed in set 0, overwriting the last value
- Read from address 10, this should be placed in set 1, overwriting the last value

Fully Associative

Initialize a cache with 8 byte cache size, 16 byte memory size, block size of 2, set size of 2, FIFO replacement strategy, fully associative placement policy, write back write policy.

- Read from address 0, this should be placed in set 0
- Read from address 2, this should be placed in set 0, confirming that there is only one single set
- Read from any address, and it will fill the cache, and when it is full it will replace the values according to the replacement policy

Set Associative

Initialize a cache with 8 byte cache size, 16 byte memory size, block size of 2, set size of 2, FIFO replacement strategy, set associative placement policy, write back write policy.

- Read from address 0, this should be placed in set 0
- Read from address 2, this should be placed in set 1
- Read from address 4, this should be placed in set 0
- Read from address 6, this should be placed in set 1
- Read from address 8, this should be placed in set 0, overwriting the first value read from address 0

Conclusion

Cache memories are a crucial component of computer systems and play a vital role in improving performance by storing frequently accessed data and instructions. I implemented a cache memory simulator in Java, which allowed me to explore and understand the various behaviors and policies that can be used to manage a cache memory.

I implemented different placement policies such as direct mapping, fully associative, and n-way set associative, and learned how each of them maps memory addresses to cache locations differently. I also implemented different replacement policies such as LRU, LFU, and FIFO, and learned how they determine which cache lines to replace when a new memory access occurs. Finally, I implemented different write policies such as write-through and write-back, and learned how they determine when to write data back to the main memory.

Implementing these behaviors and policies in Java allowed me to gain a deeper understanding of how cache memories work and how different design decisions can affect their performance. I may have encountered some challenges along the way, but overcoming them helped me to learn and grow as a programmer. Overall, working on this project was a valuable and enjoyable learning experience.

Bibliography

1. Jim Handy, The Cache Memory Book, Morgan Kaufmann; 2nd edition (January 27, 1998)
2. D. A. Patterson and J. L. Hennessy, “Computer Organization and Design: A Quantitative Approach”, 5 th edition, ed. MorganKaufmann, 2011.
3. https://www.researchgate.net/publication/319467661_A_Novel_Longest_Distance_First_Page_Replacement_Algorithm_(Kumar, 2017)
4. https://en.wikipedia.org/wiki/CPU_cache
5. https://www.geeksforgeeks.org/cache-memory-in-computer-organization/