Cache Memory Simulation

Student: Cornoc Roland Nicholas

Structure of Computer Systems Project

Technical University of Cluj-Napoca

January 15, 2024

Contents

Introduction

1.1 Context

The purpose of this project is to simulate the operation of a cache memory unit, an essential component in modern computing systems designed to reduce the time it takes to access data from the main memory. Cache memory is a small, high-speed memory placed between the CPU and the main memory, storing frequently accessed data for quick retrieval. By storing a subset of data closer to the processor, cache memory minimizes the latency involved in memory accesses, improving the overall performance of the system.

The simulation focuses on the core operations of cache memory, including placing new information (write), retrieving existing information (read), and evicting or replacing old information (cache eviction). These operations are fundamental to understanding how cache memory optimizes access times and manages data.

1.2 Objectives

The project will be implemented and will simulate the functionality of a cache memory unit. The simulation will focus on the primary operations performed by a cache memory system, such as placing new information in the cache, retrieving stored information, and evicting outdated or less frequently used data. A graphical interface will be developed to visualize the cache's behavior during these operations.

The simulation will also aim to demonstrate the efficiency improvements that cache memory provides to modern computer systems by reducing access time to frequently used data. The following key operations will be emphasized:

Types of Cache Operations and Behaviors

- Cache Hit: Retrieving data from the cache if it is already stored, demonstrating the speed benefit of caching.
- Cache Miss: Handling cases where the required data is not found in the cache and must be fetched from a slower main memory.
- Replacement Policy: Demonstrating various cache replacement strategies (e.g., Least Recently Used (LRU), First-In-First-Out (FIFO), etc.) for managing limited cache space.
- Cache Write Policies: Implementing and illustrating write strategies like Write-Through or Write-Back to manage how data updates are handled between cache and main memory.

The simulation will aim to provide a clear understanding of how cache memory contributes to overall system performance, allowing users to observe and learn about key aspects of cache operation in real-time.

Bibliographic Research

2.1 What is a Cache Memory?

Cache memory is a small, high-speed storage located between the CPU and the main memory, designed to store frequently accessed data and instructions. Its primary function is to reduce the time it takes for the processor to access data, thereby improving overall system performance. When the CPU requests data, it first checks the cache. If the requested data is present, it leads to a **cache hit**, and the CPU retrieves it quickly from the cache. If the data is not found, it results in a **cache miss**, and the data must be fetched from slower main memory. The **hit rate** refers to the percentage of times the cache provides the required data, while the **miss rate** indicates how often the cache fails to find the requested information. Efficient cache systems aim for a high hit rate, reducing delays caused by cache misses and improving overall system efficiency.

2.2 Reducing Cache Misses

Reducing the number of cache misses is crucial to maintaining high system performance. Several strategies can be employed to achieve this. **Increasing cache size** allows more data to be stored, reducing the chances of a miss. **Improving block size** can also help by fetching larger chunks of data, thereby increasing the likelihood of future cache hits. **Replacement policies** like Least Recently Used (LRU) or First-In-First-Out (FIFO) can help optimize which data to evict when the cache is full, keeping the most relevant information. Another approach involves **cache associativity**, where the mapping of data between main memory and cache becomes more flexible, decreasing conflicts that lead to misses. Prefetching techniques that load data into the cache before it's requested can also reduce cache miss rates.

2.3 Where can a block be placed in the Cache?

Cache memory is organized in such a way that data blocks from the main memory are mapped to specific locations within the cache. The strategy used to determine where a block is placed in the cache is determined by the cache's **associativity**. There are three main types of cache associativity: **Direct Mapped**, **Fully Associative**, and **Set Associative**.

2.3.1 Cache Memory Organization

- **Direct Mapped Cache**: In a direct-mapped cache, each block of data from the main memory can be placed in exactly one location (or cache line) within the cache. This strict one-to-one mapping makes direct-mapped caches simple and fast, but also increases the likelihood of **cache conflicts** where multiple memory blocks compete for the same cache line. If two blocks need to be placed in the same location, the existing block is evicted, which may lead to higher miss rates in comparison to more flexible cache organizations.

- **Fully Associative Cache**: In a fully associative cache, a block of memory can be placed in **any available cache line**, providing maximum flexibility. This type of cache minimizes conflicts since any memory block can be stored anywhere, reducing the chances of eviction due to conflicts. However, fully associative caches are more complex to implement, as they require checking every cache line during lookups, which can be slower and more resource-intensive.

- **Set Associative Cache**: A set associative cache is a middle-ground approach between direct-mapped and fully associative caches. The cache is divided into a number of **sets**, and each block from memory can be placed in any line within a specific set. For example, in an **N-way set associative cache**, a block can be placed in one of N possible lines within a set. The number of sets is typically calculated as (Block Address) MOD (Number of Sets in the Cache). This type of cache balances flexibility and performance, reducing the conflict rate seen in direct-mapped caches while being less complex than fully associative caches.

2.3.2 Associativity and Performance

The associativity of a cache impacts its performance, particularly the **miss rate**. A higher associativity generally leads to fewer cache misses because the cache can store blocks more flexibly, reducing conflicts. For example, the miss rate of a **direct-mapped cache** of size X is approximately the same as that of a **2- to 4-way set associative cache** of size X/2. This means that increasing the associativity allows for a smaller cache to perform as well as or better than a larger, direct-mapped cache.

2.4 Block Identification

In cache memory, **block identification** refers to the process of determining whether a requested data block from the **main memory** is present in the cache (a hit) or not (a miss). The method for identifying a block in the cache depends on the cache's organization—whether it's **direct mapped**, **set associative**, or **fully associative**. Each of these organizations handles block identification differently, with various trade-offs between complexity, speed, and efficiency.

2.4.1 Set Associative Cache

In a set associative cache, the cache is divided into multiple sets, and each set can store multiple blocks. When identifying a block, the cache uses the block address to determine the set in which the block might reside by applying a modulo operation:

(Block Address) MOD (Number of Sets).

Once the appropriate set is determined, the cache compares the tag of the incoming block to the tags of all blocks in that set. If a matching tag is found, it’s a cache hit, and the data is fetched from the cache. Otherwise, it's a cache miss, and the block is fetched from main memory.

Advantages of Set Associative Cache

- **Higher Hit Rate for the Same Cache Size**: Set associative caches reduce the chances of conflicts because multiple blocks can reside in each set. This improves the overall hit rate compared to direct-mapped caches of the same size.
- **Fewer Conflict Misses**: By allowing more than one block per set, set associative caches minimize the number of conflict misses, which occur when two blocks compete for the same location in the cache.

Disadvantages of Set Associative Cache

- **N Comparators vs. 1**: In an N-way set associative cache, N comparators are required to check all potential cache lines within a set, increasing complexity compared to a direct-mapped cache, which only needs one comparator.
- **Extra MUX Delay for Data**: The process of selecting the correct cache line introduces extra delay, as the data can only be accessed after the hit/miss decision and set selection have been made.
- **Data Comes After Hit/Miss Decision**: Unlike direct-mapped caches, where the data is immediately available and assumed to be a hit until proven otherwise, set associative caches only provide data after confirming whether there's a hit or miss. This can slightly increase access latency.

2.4.2 Fully Associative Cache

In a **fully associative cache**, there are no restrictions on where a block from memory can be placed. Any block can be stored in any available cache line, giving the system complete flexibility in managing cache space. When a block is requested, the cache searches all of its lines simultaneously to determine whether the block is present. This process, known as parallel comparison, is enabled by the use of specialized hardware, such as **content addressable memory (CAM)**, which allows the cache to compare the tags of all cache lines at once.

This organization eliminates the issue of **conflict misses**—a common problem in direct mapped caches—since any block can be stored anywhere in the cache. However, the flexibility of fully associative caches comes at a cost. The hardware required for parallel tag comparisons is more complex, and the overall system can be slower due to the increased time needed to search through every cache line simultaneously.

Advantages of Fully Associative Cache

- **No Conflict Misses**: Since any block can be placed anywhere in the cache, there are no conflict misses, which occur when two blocks need to be stored in the same cache line.
- **Maximum Flexibility**: The cache can make efficient use of all available space, as there are no restrictions on where data can be stored.
- **Improved Cache Utilization**: The likelihood of cache hits increases since there is no predefined mapping of blocks to specific cache lines.

Disadvantages of Fully Associative Cache

- **High Hardware Complexity**: Parallel comparisons across all cache lines require sophisticated hardware, making fully associative caches more expensive and resource intensive.
- **Slower Lookup Time**: While fully associative caches can eliminate conflict misses, the process of comparing every cache line's tag in parallel can introduce delays compared to simpler cache designs.
- **Increased Power Consumption**: The complexity of searching all cache lines simultaneously leads to higher power usage, which can be a drawback in energy-sensitive systems.

2.4.3 Direct Mapped Cache

In a **direct mapped cache**, each block from main memory is assigned to a single, specific location within the cache. This means that for any given memory address, there is only one possible cache line where its corresponding block can be placed. The identification process is straightforward: when a block is requested, the cache looks up the corresponding line directly and checks whether the data stored in that line matches the requested block. If the data is present, it results in a **cache hit**; if not, it's a **cache miss**, and the block must be retrieved from the slower main memory and placed into the cache.

Because each block can only occupy one fixed location, the lookup and retrieval process is simple and fast. However, this strict placement policy can lead to **conflict misses**, where multiple blocks that map to the same cache line continually overwrite each other, even if there's unused space elsewhere in the cache. Despite its simplicity, direct mapped caches are susceptible to performance issues when multiple frequently accessed blocks map to the same line.

Advantages of Direct Mapped Cache

- **Simplicity and Speed**: The fixed placement of each block allows for quick lookups, as there's only one possible cache line to check.
- **Low Complexity**: Since only one comparison is needed to check for a match, the hardware design is simpler and less costly.
- **Fast Access**: The cache can optimistically assume a hit and provide the data immediately, improving access times for hits.

Disadvantages of Direct Mapped Cache

- **Higher Conflict Miss Rate**: Blocks that map to the same line frequently replace each other, leading to more misses and reduced performance.
- **Inflexibility**: The rigid mapping of blocks to specific cache lines limits the cache's ability to efficiently store frequently accessed data.
- **Frequent Replacements**: In workloads where multiple blocks conflict for the same cache line, the frequent eviction of useful data increases the number of misses.

2.5 Block replacement Policy

When a cache miss occurs and the cache is full, the system must decide which existing block should be replaced to make room for the new block. This decision is critical for cache performance, particularly in associative caches, where multiple blocks can be stored in any location or set, and thus, a block replacement policy is required to manage the cache efficiently. Below are the common block replacement strategies used in associative caches.

A. Direct Mapped Cache

In a direct mapped cache, block replacement is straightforward. Since each block can only be placed in one specific location, there is no need for a replacement algorithm. The new block will simply replace the existing block in the same cache line.

B. Associative Caches

In fully associative and set associative caches, where blocks can be placed in multiple locations, a replacement algorithm is needed to decide which block should be evicted when a new block needs to be loaded. Here are several widely used replacement algorithms:

There are several widely used block replacement algorithms in cache memory systems, each with its own set of strengths and weaknesses.

❖ **Least Recently Used (LRU)**

The **Least Recently Used (LRU)** policy evicts the block that has not been used for the longest time, operating on the assumption that blocks accessed recently are more likely to be needed again soon. This strategy is particularly effective in workloads where data that was recently accessed tends to be reused. In small caches, LRU can be implemented with relative simplicity by tracking usage with basic bits, especially in a 2-way set associative cache. However, in fully associative caches, LRU requires maintaining a list or more complex data structure to keep track of all block references. While LRU offers a good balance between performance and complexity, it can become costly in terms of overhead in larger caches due to the need to keep track of each block's usage history.

❖ **First-In, First-Out (FIFO)**

The **First-In, First-Out (FIFO)** policy replaces the block that has been in the cache the longest, without considering how often or how recently the block was accessed. This method is easy to implement and requires minimal bookkeeping. FIFO is often organized using a simple round-robin or circular buffer system, where blocks are replaced in the order they were loaded into the cache. While FIFO is straightforward and efficient in terms of complexity, it may not always result in optimal performance because it does not account for how often a block is used, leading to potential inefficiencies in data-heavy applications.

❖ **Least Frequently Used (LFU)**

The **Least Frequently Used (LFU)** policy focuses on evicting the block that has been accessed the least frequently. This algorithm assumes that blocks accessed less often are less likely to be needed in the future. Each block has a counter that increments with every access, and the block with the lowest counter is replaced when needed. LFU is particularly useful in situations where frequently accessed data should remain in the cache for as long as possible. However, it can also lead to **cache pollution**, where older blocks with high access counts stay in the cache even when newer data, which might be more relevant, needs to be stored. As a result, LFU may not always be the best choice for every workload.

❖ **Most Recently Used (MRU)**

In contrast, the **Most Recently Used (MRU)** policy replaces the block that was most recently used. This strategy is advantageous in specific types of workloads, such as stack-based algorithms, where the most recently accessed data is less likely to be needed again soon. While MRU can outperform LRU in certain cases, particularly in specialized applications, it tends to perform worse in general-purpose computing environments, where more predictable patterns of data reuse make LRU a better choice.

❖ **Random Replacement**

The **Random Replacement** policy, as the name suggests, selects a block to evict randomly, without regard to how often or how recently the block was accessed. Despite its simplicity, random replacement has been shown to perform surprisingly well in various real-world scenarios, often coming close to the performance of LRU. Its major advantage is its simplicity, as it requires no complex tracking of usage patterns or bookkeeping. However, because it does not adapt to the access patterns of the workload, random replacement can

underperform in situations where there is a high degree of data locality, leading to more frequent cache misses.

Remark: Comparison of Block Replacement Policies

- **LRU** and **LFU** are more **intelligent** policies, as they consider access patterns, but they are also more **complex** and resource-intensive to implement.
- **FIFO** and **Random** are **simpler** but may result in suboptimal performance for certain workloads.
- **MRU** is effective in specific cases but generally does not outperform LRU or LFU in most typical scenarios.

2.6 Write Policy

In cache memory systems, **write policies** dictate how data modifications in the cache are handled with respect to the main memory. When a CPU writes data to a location stored in the cache, the system needs to determine how and when that data is updated in the main memory to maintain consistency.

2.6.1 Write-Through and Write-Back

The two primary write policies are **Write-Through** and **Write-Back**, each with distinct trade-offs in terms of speed, complexity, and data consistency.

2.6.1.1 Write-Through Policy

In the **Write-Through** policy, every time data is written to the cache, the same data is immediately written to the main memory. This approach ensures that the cache and main memory are always synchronized, providing high data integrity. If a cache block is modified, there is no need to worry about inconsistency, as the changes are reflected in main memory instantly.

While **Write-Through** ensures consistency, it introduces higher memory traffic, as every write operation generates a corresponding write to the slower main memory. This can reduce overall performance, especially in workloads with frequent writes. To mitigate this, many systems use a write buffer to temporarily store write operations, allowing the CPU to continue without waiting for each memory write to complete.

2.6.1.2 Write-Back Policy

In contrast, the **Write-Back** policy only updates the main memory when the cache block containing the data is replaced or evicted from the cache. When data is written to the cache, it is marked as "dirty," indicating that it has been modified. The dirty bit signals that the block must be written back to main memory only when it is removed from the cache, rather than on every write operation. This reduces memory traffic, as multiple writes to the same cache block can be handled without involving main memory until absolutely necessary.

Write-Back improves performance by reducing the number of slow memory writes. However, it complicates cache management since the system must keep track of which blocks are dirty and ensure that these are properly written back when replaced. Additionally, in the case of a system failure, data that has been modified but not yet written back to memory may be lost, leading to potential consistency issues.

2.6.2 Write Allocate vs. No-Write Allocate

Both **Write-Through** and **Write-Back** policies can be paired with either a Write Allocate or No-Write Allocate strategy.

2.6.2.1 Write Allocate

In **Write Allocate**, when a write miss occurs (i.e., when the data being written is not in the cache), the cache first loads the block from main memory into the cache and then performs the write operation. This strategy assumes that future writes to the same block may occur, so it is beneficial to load the block into the cache.

2.6.2.2 No-Write Allocate

In **No-Write Allocate**, the data is directly written to the main memory without loading the block into the cache on a write miss. This is often used in conjunction with the Write-Through policy, where the main memory is always updated immediately, making cache allocation less critical for writes.

Analysis

3.1 Functional Analysis

The **cache simulation program** is designed to provide an interactive and visual representation of how cache memory functions in a computer system. It aims to replicate real-world scenarios to help users understand key concepts such as placement, retrieval, and replacement of information in a cache. By offering a customizable platform, users can modify various cache parameters and observe how these changes influence performance.

The program's primary functionality revolves around three core operations. First, it simulates the placement of new information in the cache, allowing users to see how memory addresses are mapped to cache blocks. Second, it enables the retrieval of information, indicating whether a given memory address results in a **cache hit** or a **miss**. Finally, the program handles replacement of information in cases of cache misses, using policies such as **FIFO**, **LRU**, **Random**, **MRU** or **LFU**.

In addition to these basic functions, the simulation includes options to choose the type of cache that the end user desires to use, such as **Fully Associative (FA)** or **Direct Mapped Cache.** These configurations allow the user to experiment with how data is handled during write operations, providing deeper insights into memory management systems.

3.2 Features and Use Cases

3.2.1 Cache Configurations

The program offers significant flexibility by allowing users to select **different cache configurations**. Users can choose between various levels of associativity, including **Fully Associative (FA), 2-Way Associative, 4-Way Associative**, or **Direct Mapped Cache**. These options ensure that the program caters to a wide range of scenarios, from highly flexible to strictly mapped cache systems.

Additionally, the user can define the **cache size** and **memory size**, both constrained to powers of 2. This customization helps in simulating different system architectures. The user can also specify the number of **offset bits**, which determines the data granularity within a cache block. These parameters enable precise control over the simulation setup and allow for tailored experimentation.

3.2.2 Replacement Policies

The program supports three **replacement policies**: **FIFO (First In, First Out), LRU (Least Recently Used), and Random**. **FIFO** ensures that the oldest block is replaced first, **LRU** prioritizes the block that has not been accessed for the longest time, and **Random** selects a block at random. Each policy has distinct implications for cache performance, and the program enables users to visualize their effects during simulation.

3.3 Visualization and Interaction

3.3.1 Address Mapping and Placement

The program allows users to input **memory addresses**, which are then mapped to specific **cache blocks**. For each address, the simulation calculates the corresponding **set/index, tag**, and **offset bits**. The placement of the address in the cache is visually represented, highlighting the block where the data is stored. Users can observe whether the address results in a cache hit or miss, and in case of a miss, the program shows if a block was replaced based on the selected **replacement policy**.

3.3.2 Simulation Process

The simulation operates step-by-step, making it easy for users to track each operation. When a series of addresses are provided, the program processes each address in sequence, displaying:

- Whether the operation is a read or write.
- The calculated cache set/index and tag for the address.
- The state of the cache after the operation.
- This detailed visualization helps users understand the inner workings of cache memory, such as data flow and block replacement.

3.3.3 Insights on Performance

To provide actionable insights, the program calculates key performance metrics such as the **cache hit rate, miss rate**, and **replacement count**. These metrics give users a quantitative understanding of how different configurations and policies affect overall cache performance.

3.4 Use Cases and Applications

This program is ideal for several use cases. As an **educational tool**, it offers students and educators a hands-on way to explore **cache memory** concepts. Visualizing data placement and replacement makes abstract concepts easier to understand, enhancing learning outcomes.

For performance testing, developers can use the program to experiment with various cache configurations to identify the most effective strategies for specific workloads. For instance, they can compare the performance of **LRU** and **FIFO** policies in scenarios with **high cache contention**.

The simulation can also act as a **debugging aid** for system architects. By visualizing **memory access patterns** and **cache behavior**, architects can identify inefficiencies in **memory management** and optimize system design.

3.5 Scalability and Extensibility

The program is designed to handle a wide range of **configurations**, making it **scalable** for both small and large memory systems. Its modular structure allows for easy addition of new features. For example, future versions could include support for **multi-level caches**, **prefetching algorithms**, or advanced metrics like **average memory access time**.

Design

4.1 Overview

The **Cache Simulator** application is a software tool developed to visually and interactively simulate the functionality and operations of a CPU cache. Designed primarily as an educational aid, this application allows users to explore and understand the core principles of caching by configuring various parameters and observing how cache behavior changes in response. Users can set attributes such as **memory size, cache size, block size, mapping techniques**, and **replacement policies**, and then simulate cache accesses step by step to gain insights into performance metrics such as **cache hits, misses**, and **evictions**.

The application is implemented using **Python** for its backend logic and **PyQt5** for building a **graphical user interface (GUI)** that is both modern and intuitive. By combining a robust simulation engine with a visually appealing interface, the **Cache Simulator** makes complex CPU caching concepts accessible to students, educators, and technology enthusiasts.

4.2 Features

1. User-Friendly GUI

The application features a clean and responsive graphical interface built with PyQt5. Users can easily interact with the simulator through organized input fields, buttons, and tables.

Tooltips, error messages, and visual feedback are incorporated to improve usability and ensure smooth navigation through the interface.

2. Customizable Parameters

Users have full control over key cache and memory settings, enabling them to test different configurations and observe their impact on cache behavior. The configurable parameters include:

- **Memory Size (in bytes):** The total size of the addressable memory space.
- **Cache Size (in bytes):** The storage capacity of the cache.
- **Block Size (in bytes):** The size of individual blocks (or cache lines) within the cache.
- **Mapping Techniques:**
    - Direct Mapping
    - Fully Associative
    - (Future support for n-way Set Associative)

- **Replacement Policies:**
    - LRU (Least Recently Used)
    - FIFO (First-In-First-Out)
    - RANDOM (Random Replacement)
    - LFU (Least Frequently Used)
    - MRU (Most Recently Used)

3. Simulation Capabilities

Step-by-Step Simulation: Users can simulate memory accesses incrementally, observing changes to the cache state after each step.

- **Visualization of Cache Behavior:**
    - Cache hit/miss results are clearly displayed.
    - Tag, index, and offset calculations are shown for each memory address.
    - The cache table updates dynamically to reflect its current state.
- **History of Results:** A comprehensive log of all simulation steps is maintained for review.

4. Random Address Generation

The application includes a feature for generating random memory addresses. This allows users to test cache behavior under various workloads without manually entering address sequences.

5. Modern Design

- The interface adopts a stylish and professional look with:
    - Rounded buttons and input fields.
    - A subtle gray and blue color scheme for improved visual contrast.
    - Clear and readable fonts (e.g., 'Segoe UI', Arial) for better text presentation.
    - Alternating row colors in tables for enhanced readability.

4.3 Architectural Design

1. Core Classes

a. CacheSimulator Class

This class forms the backbone of the application and handles the core cache simulation logic. Its responsibilities include:

- **Initialization:** Setting up cache parameters, memory structures, and data fields.
- **Address Translation:** Breaking down memory addresses into tag, index, and offset components based on cache configurations.
- **Mapping Techniques:**
    - **Direct Mapping:** Maps each memory block to a specific cache line.
    - **Fully Associative:** Allows any memory block to be stored in any cache line.
- **Replacement Policies:**
    - LRU: Evicts the least recently used block.
    - FIFO: Removes the oldest block in the cache.
    - RANDOM: Evicts a randomly selected block.
    - LFU: Removes the least frequently accessed block.
    - MRU: Evicts the most recently accessed block.
- **Cache Operations:**
    - Determining whether a memory access results in a cache hit or miss.
    - Handling cache misses by loading new data and evicting old data if necessary.
    - Tracking statistics such as total hits, misses, and evictions.

b. CacheSimulatorApp Class

This class manages the graphical user interface and serves as the intermediary between the user and the simulation logic. Key responsibilities include:

- **Input Validation:** Ensures that user-provided parameters are valid, such as positive values for memory and cache sizes and valid address ranges.
- **Dynamic GUI Updates:**
    - Populating and updating tables (e.g., cache table, instruction table).
    - Displaying hit/miss results and maintaining a log of simulation steps.
- **User Interaction:** Responding to button clicks and triggering corresponding actions in the CacheSimulator class.

2. UI Components

a. Input Fields

- Used to collect parameters such as memory size, cache size, block size, mapping technique, replacement policy, and memory address sequence.
- Input validation is enforced to ensure data integrity.

b. Buttons

- **Simulate:** Starts the simulation based on user-defined parameters.

- **Generate Random Addresses:** Creates a random sequence of memory addresses for testing.
- **Next:** Steps through the simulation one memory access at a time.

c. Tables

- **Cache Table:** Displays cache content, including fields such as index, valid bit, tag, and data.
- **Instruction Table:** Shows tag, index, and offset calculations for each memory address.

d. Text Areas

- **Result Textbox:** Displays real-time feedback on cache hit/miss results and overall performance metrics.
- **History Textbox:** Logs all simulation steps, enabling users to trace the simulation process.
- **Hit/Miss Summary:** Provides a cumulative summary of cache hits and misses.

4.4 Simulation Logic

The simulation logic is central to the application and encompasses several key operations:

1. Address Translation

- Memory addresses are divided into three components:
    - **Tag:** Identifies the unique memory block.
    - **Index:** Specifies the cache line where the block may reside (used in Direct Mapping).
    - **Offset:** Identifies the specific byte within a block.

2. Cache Access

- **Hit/Miss Detection:**
    - A hit occurs if the requested memory block is already present in the cache.
    - A miss occurs if the block is not found, triggering a data load from memory.
- **Replacement Policies:** Determines which cache line to evict during a miss.

3. Dynamic Updates

- The GUI updates in real time during each simulation step:
    - The cache table reflects the current state of the cache.
    - The instruction table displays calculations for the current memory address.

- The result and history textboxes are updated with feedback and logs.

4.5 Error Handling

To ensure a seamless user experience, the application incorporates robust error handling mechanisms:

- **Invalid Input:** Displays descriptive error messages for invalid parameter entries (e.g., negative values, empty fields).
- **Memory Access Out of Range:** Prevents simulation of addresses outside the specified memory size.
- **Cache Overflow:** Ensures proper eviction policies are followed when the cache is full.

4.6 Extensibility

The application is designed with future enhancements in mind, allowing developers to easily expand its functionality. Possible extensions include:

- **Support for Additional Mapping Techniques:** Adding n-way Set Associative Mapping.
- **Write Policies:** Implementing write-through and write-back policies.
- **Advanced Visualizations:** Including graphical representations of cache hierarchy and timing diagrams.
- **Custom Replacement Policies:** Allowing users to define their own eviction algorithms.

4.7 Design Conclusion

The Cache Simulator application bridges the gap between theoretical concepts and practical understanding of CPU caching. By combining a powerful simulation engine with an engaging and user-friendly interface, it serves as an effective educational tool for students, educators, and professionals. Its modular architecture and extensible design ensure long-term usability and adaptability for evolving needs in teaching and learning about computer architecture.

Implementation

The **Cache Simulator Application** is a sophisticated tool designed to emulate the behavior of a computer memory cache. It provides insights into how different cache configurations, mapping techniques, and replacement policies impact performance. The application is built around two core classes, **CacheSimulator** and **CacheSimulatorApp**, each playing distinct roles in the simulation.

5.1 Methods in the CacheSimulator Class

The **CacheSimulator** class is the core of the application, encapsulating all logic and operations related to simulating a cache system. This class handles the intricate details of cache memory, including mapping techniques, replacement policies, and performance metrics. Its methods are carefully crafted to ensure accurate and efficient simulation of memory accesses, enabling users to understand how different configurations affect cache behavior.

- **__init__(self, memory_size, cache_size, block_size, mapping_technique, replacement_policy)**

The **__init__ method** is the constructor of the class and is responsible for initializing all the necessary parameters and data structures. When an instance of the class is created, this method takes in several critical arguments, such as **memory size, cache size, block size, mapping technique,** and **replacement policy**.

These parameters define the overall structure and behavior of the cache. For instance, the memory size specifies the total addressable space, while the cache size and block size determine how memory is divided and accessed. Based on these inputs, the method allocates storage for the cache and initializes auxiliary data structures, such as tag arrays, metadata for tracking access patterns, and counters for hits and misses.

It also validates the provided parameters, ensuring that values like memory size and cache size are powers of two and that the block size is appropriate for the selected configuration.

- **translate_address(self, address)**

The **translate_address** method plays a crucial role in the simulation by decomposing memory addresses into their constituent components: **tag, index,** and **offset**. This method takes a memory address as input and, using the cache configuration, calculates the corresponding

index and offset.

The tag is derived from the higher-order bits of the address, while the index and offset are determined based on the block size and the number of cache sets. These components are essential for locating data in the cache and determining whether a particular address results in a hit or a miss.

For instance, in a direct-mapped cache, the index specifies the cache set, while the tag is compared against the stored tag in that set to verify if the data is present.

- **access_cache(self, address)**

The **access_cache** method is the heart of the simulation, handling memory accesses and determining the outcome of each access. When a memory address is accessed, this method first calls translate_address to break it down into **tag, index**, and **offset** components.

It then checks the cache set corresponding to the index to see if the tag matches any of the stored tags. If a match is found, it signifies a cache hit, and the corresponding counter is incremented. On a miss, the method invokes handle_miss to load the requested block into the cache.

This process involves not only fetching the data but also deciding which block to evict, depending on the replacement policy. The method also updates internal data structures, such as access timestamps or usage counters, to reflect the current state of the cache.

- **handle_miss(self, address)**

The **handle_miss** method deals specifically with cache misses, ensuring that the requested data is loaded into the cache while adhering to the replacement policy. This method is particularly important in scenarios where the cache is full, as it determines which block to evict to make room for new data.

For example, if the Least Recently Used (LRU) policy is selected, the method identifies the block that has not been accessed for the longest time and replaces it. Similarly, for the Random policy, it selects a block at random. By abstracting this functionality into a separate method, the class maintains a modular design, making it easier to extend or modify the simulation.

- **apply_replacement_policy(self, index)**

The **apply_replacement_policy** method is responsible for implementing the selected policy for evicting blocks from the cache. Depending on the user's configuration, it may execute one of several strategies, such as **LRU**, **First-In-First-Out (FIFO), Most Recently Used (MRU),** or **Least Frequently Used (LFU)**.

This method typically operates on a specific cache set, using metadata like access timestamps or usage counters to identify the target block. For instance, in the case of LFU, the method scans the cache set to find the block with the lowest access frequency and marks it for replacement. This flexibility allows users to experiment with different policies and observe how they impact cache performance under various workloads.

- **calculate_statistics(self)**

The **calculate_statistics** method is the final piece of the puzzle, aggregating and presenting performance metrics to the user. This method computes key statistics, including the total number of **hits** and **misses**, the **hit rate**, and the **miss rate**. These metrics provide valuable insights into the efficiency of the cache configuration and highlight areas for improvement.

For example, a high miss rate might suggest that the cache size is too small or that the chosen replacement policy is suboptimal for the given workload. The method ensures that these statistics are updated in real time during the simulation, offering users a clear view of how their choices affect performance.

Conclusion

The methods within the **CacheSimulator** class form the backbone of the application, implementing all the logic required to simulate a cache system. From initializing parameters and breaking down memory addresses to managing cache misses and applying replacement policies, each method is designed to work seamlessly with the others. Together, they provide a comprehensive and flexible framework for exploring the complexities of cache memory.

5.2 Methods in the CacheSimulatorApp Class

The **CacheSimulatorApp** class serves as the interface layer of the application, providing users with the tools and interactions necessary to engage with the cache simulation. It is built on **PyQt5**, a powerful Python library for creating Graphical User Interfaces (GUIs). This class bridges the gap between the user and the underlying simulation logic housed in the **CacheSimulator** class. By managing inputs, outputs, and events, the class ensures a seamless and user-friendly experience.

Each method in the **CacheSimulatorApp** class is designed to handle a specific aspect of the user interaction process, from initializing the interface to validating inputs and presenting results. Below is a detailed breakdown of its key methods.

- **__init__(self)**

This method serves as the constructor, where all GUI components are initialized and laid out. When the application is launched, this method creates input fields for parameters like memory size, cache size, block size, mapping techniques, and replacement policies. Each of these fields is carefully labeled to ensure clarity for the user, with tooltips providing additional guidance on acceptable input ranges and formats.

- **validate_inputs(self)**

Another critical method, **validate_inputs**, ensures that all data entered by the user meets the required criteria. This step is vital to prevent errors during the simulation and to provide immediate feedback if any issues are detected. For example, the method checks that the memory size and cache size are positive integers and that both are powers of two. It also verifies that the block size is smaller than or equal to the cache size and that a valid mapping technique and replacement policy are selected.

If any of these conditions are not met, an error message is displayed to the user, either through a popup window or an inline indicator. For instance, if the block size exceeds the cache size, the application shows a warning stating, "Block size must not exceed cache size." This mechanism not only ensures data integrity but also enhances the user experience by providing clear and actionable feedback.

- **run_simulation(self)**

Once the inputs are validated, the **run_simulation** method comes into play. This method acts as the main driver of the application. It collects user inputs, initializes the simulation engine, and processes memory accesses based on the provided parameters. First, the method retrieves values from the input fields, such as the memory size, cache size, and selected mapping technique. These values are then passed to the **CacheSimulator** class to create an instance of the simulation engine.

The simulation engine operates on a sequence of memory addresses, which may either be provided by the user or generated randomly within the specified memory size. For each memory address, the access_cache method in the simulation engine determines whether the address results in a cache hit or miss. This information is then returned to the GUI, where the cache state and performance metrics are updated in real time

- **update_ui(self, statistics, cache_state)**

Updating the interface is handled by the **update_ui** method, which refreshes various components of the application to reflect the current state of the simulation. The cache state is displayed in a table format, with each row representing a cache block and columns showing details such as the tag, index, and data content of the block. This table is updated dynamically as the simulation progresses.

Recently accessed blocks may be highlighted to help users visualize the flow of data, with hits and misses distinguished using color-coded indicators—blue for hits and red for misses. Alongside the cache state, the method also updates performance metrics, such as the total number of hits and misses, the hit rate, and the miss rate.

These metrics are presented in a separate section of the interface, providing users with a clear understanding of how the cache is performing under the given parameters.

- **reset_ui(self)**

For cases where users wish to restart the simulation or adjust parameters, the **reset_ui** method offers a convenient way to clear all inputs and restore the application to its initial state. This method ensures that all input fields are emptied, and any displayed results or statistics are cleared.

Internal attributes, such as counters and data structures used to track the simulation state, are also reset. Before performing these actions, the application may prompt the user for confirmation to prevent accidental data loss.

This feature is particularly useful in educational settings, where users may want to experiment with different configurations and observe how the cache's performance changes.

Conclusion

The methods within the **CacheSimulatorApp** class are carefully designed to provide a cohesive and engaging user experience. From initializing the interface and validating inputs to running the simulation and presenting results, each method plays a crucial role in making the application both functional and intuitive.

By combining robust backend logic with a polished frontend interface, the **CacheSimulatorApp** class effectively bridges the gap between complex simulation algorithms and user accessibility.

Testing & Validation

6.1 Testing: Direct Mapping with 10 Random Memory Addresses

In this section, we document the results of testing the cache simulator with a specific configuration and a set of memory addresses. The aim is to analyze how the cache performs with direct mapping under a constrained memory and cache size.

Test Configuration

The test is conducted using the following parameters:

- **Memory size**: 256 bytes
- **Cache size**: 32 bytes
- **Block size**: 4 bytes
- **Mapping technique**: Direct Mapping

The memory address sequence used for this test is: 135, 45, 170, 62, 254, 33, 108, 99, 238, 33

Address Translation and Results

For each address, we calculate the **tag**, **index**, and **offset** based on the configuration and analyze whether the cache access results in a hit or a miss.

Memory Address: 135 (0x87)

- **Binary Representation**: 10000111
- **Index Calculation**: (135 // 4) % 8 = 33 % 8 = 1
- **Tag**: 33 // 8 = 4
- Result: **Miss** (Cache line 1 was empty).

Memory Address: 45 (0x2D)

- **Binary Representation**: 00101101
- **Index Calculation**: (45 // 4) % 8 = 11 % 8 = 3
- **Tag**: 11 // 8 = 1
- Result: **Miss** (Cache line 3 was empty).

Memory Address: 170 (0xAA)

- **Binary Representation**: 10101010
- **Index Calculation**: (170 // 4) % 8 = 42 % 8 = 2

- **Tag**: 42 // 8 = 5
- Result: **Miss** (Cache line 2 was empty).

Memory Address: 62 (0x3E)

- **Binary Representation**: 00111110
- **Index Calculation**: (62 // 4) % 8 = 15 % 8 = 7
- **Tag**: 15 // 8 = 1
- Result: **Miss** (Cache line 7 was empty).

Memory Address: 254 (0xFE)

- **Binary Representation**: 11111110
- **Index Calculation**: (254 // 4) % 8 = 63 % 8 = 7
- **Tag**: 63 // 8 = 7
- Result: **Miss** (Cache line 7 was overwritten).

Memory Address: 33 (0x21)

- **Binary Representation**: 00100001
- **Index Calculation**: (33 // 4) % 8 = 8 % 8 = 0
- **Tag**: 8 // 8 = 1
- Result: **Miss** (Cache line 0 was empty).

Memory Address: 108 (0x6C)

- **Binary Representation**: 01101100
- **Index Calculation**: (108 // 4) % 8 = 27 % 8 = 3
- **Tag**: 27 // 8 = 3
- Result: **Miss** (Cache line 3 was overwritten).

Memory Address: 99 (0x63)

- **Binary Representation**: 01100011
- **Index Calculation**: (99 // 4) % 8 = 24 % 8 = 0
- **Tag**: 24 // 8 = 3
- Result: **Miss** (Cache line 0 was overwritten).

Memory Address: 238 (0xEE)

- **Binary Representation**: 11101110
- **Index Calculation**: (238 // 4) % 8 = 59 % 8 = 3
- **Tag**: 59 // 8 = 7

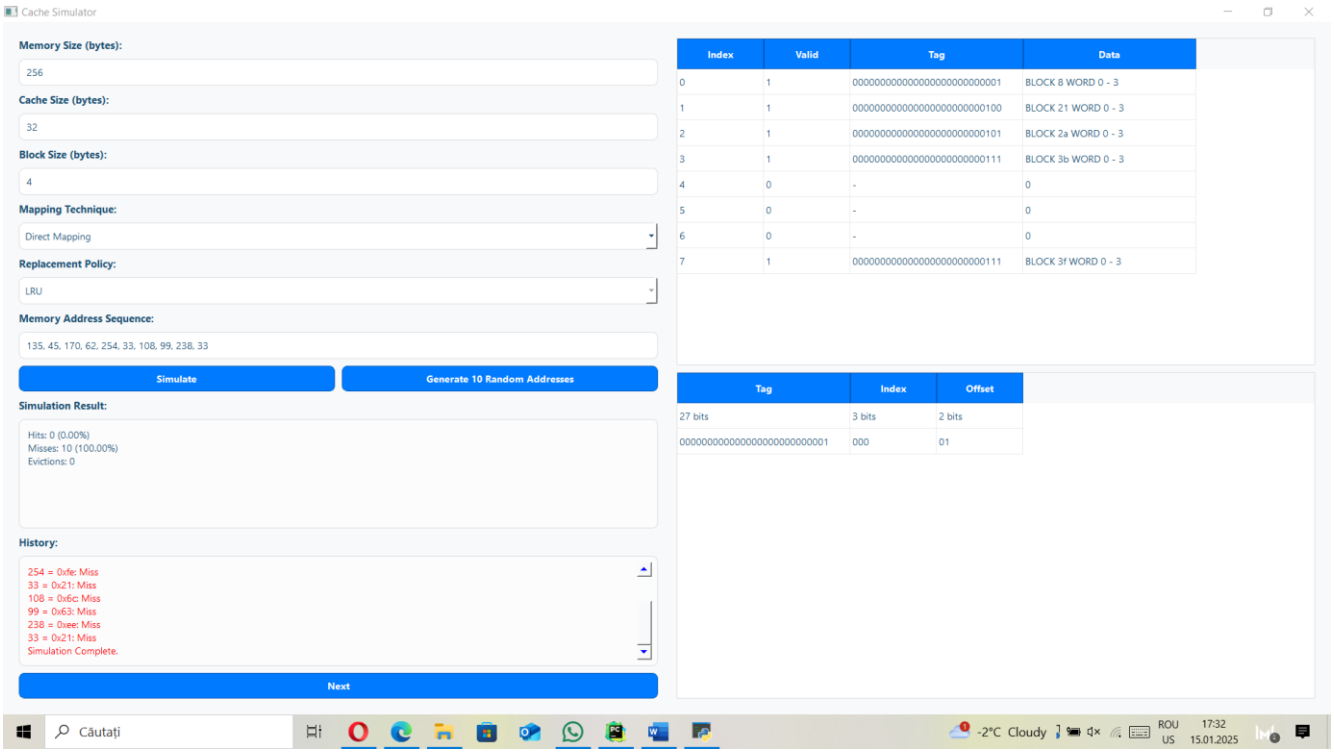
- Result: **Miss** (Cache line 3 was overwritten).

Memory Address: 33 (0x21)

- **Binary Representation**: 00100001
- **Index Calculation**: (33 // 4) % 8 = 8 % 8 = 0
- **Tag**: 8 // 8 = 1
- Result: **Miss** (Cache line 0 was overwritten).

Summary of Results

- **Hits**: 0 (0.00%)
- **Misses**: 10 (100.00%)
- **Evictions**: 0



Analysis

Despite multiple accesses to addresses such as 33, there were no hits. This outcome is expected, given the direct mapping configuration, which restricts a single cache line to store data for specific indices only. Furthermore, the frequent overwriting of cache lines (as seen with addresses like 238 and 33) highlights the limitation of direct mapping in handling high spatial locality or repeated accesses.

6.2 Testing: Fully Associative Mapping with Random Replacement Policy

In this section, we document the results of testing the cache simulator configured for fully associative mapping with a random replacement policy. This test explores how the random replacement strategy impacts cache performance when there are more memory addresses than cache blocks.

Test Configuration

The test is conducted with the following parameters:

- **Memory size**: 256 bytes
- **Cache size**: 32 bytes
- **Block size**: 4 bytes
- **Mapping technique**: Fully Associative
- **Replacement policy**: Random

The memory address sequence used for this test is: 36, 255, 234, 252, 167, 155, 120, 16, 100, 51

Address Translation and Results

In fully associative mapping, any block can occupy any cache line. For each address, we determine whether the access results in a hit or a miss. If the cache is full and a new block needs to be loaded, a block is randomly chosen for eviction.

Memory Address: 36 (0x24)

- **Binary Representation**: 00100100
- **Block Number**: 36 // 4 = 9
- Result: **Miss** (Cache was empty; block 9 was loaded into the cache).

Memory Address: 255 (0xFF)

- **Binary Representation**: 11111111
- **Block Number**: 255 // 4 = 63
- Result: **Miss** (Block 63 was loaded into the cache).

Memory Address: 234 (0xEA)

- **Binary Representation**: 11101010
- **Block Number**: 234 // 4 = 58
- Result: **Miss** (Block 58 was loaded into the cache).

Memory Address: 252 (0xFC)

- **Binary Representation**: 11111100
- **Block Number**: 252 // 4 = 63
- Result: **Hit** (Block 63 was already in the cache).

Memory Address: 167 (0xA7)

- **Binary Representation**: 10100111
- **Block Number**: 167 // 4 = 41
- Result: **Miss** (Block 41 was loaded into the cache; no eviction occurred as space was available).

Memory Address: 155 (0x9B)

- **Binary Representation**: 10011011
- **Block Number**: 155 // 4 = 38
- Result: **Miss** (Block 38 was loaded into the cache).

Memory Address: 120 (0x78)

- **Binary Representation**: 01111000
- **Block Number**: 120 // 4 = 30
- Result: **Miss** (Block 30 was loaded into the cache).

Memory Address: 16 (0x10)

- **Binary Representation**: 00010000
- **Block Number**: 16 // 4 = 4
- Result: **Miss** (Block 4 was loaded into the cache).

Memory Address: 100 (0x64)

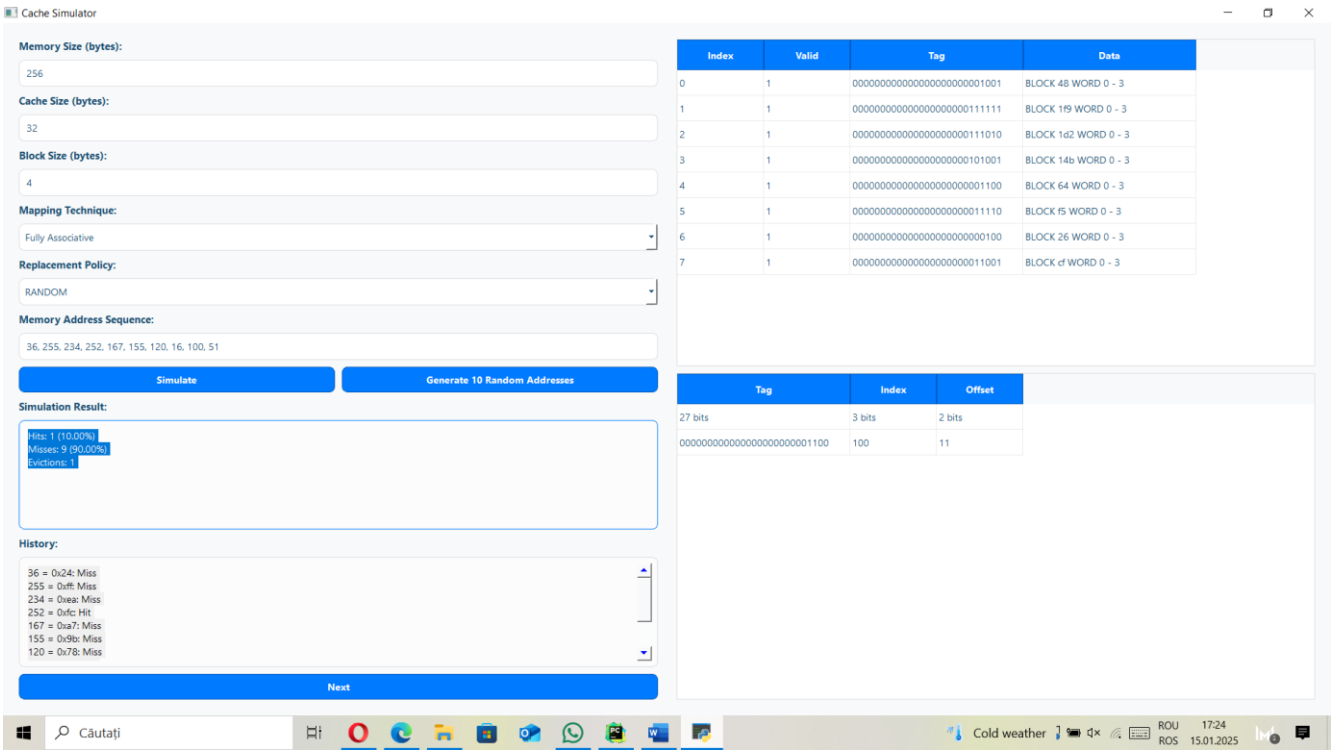
- **Binary Representation**: 01100100
- **Block Number**: 100 // 4 = 25
- Result: **Miss** (Block 25 was loaded into the cache, and one block was randomly evicted).

Memory Address: 51 (0x33)

- **Binary Representation**: 00110011
- **Block Number**: 51 // 4 = 12
- Result: **Miss** (Block 12 was loaded into the cache, and another block was randomly evicted).

Summary of Results

- **Hits**: 1 (10.00%)
- **Misses**: 9 (90.00%)
- **Evictions**: 1



Analysis

This test demonstrates the behavior of the **random replacement policy** in fully associative mapping. The policy arbitrarily selects a cache block for eviction when the cache is full. Despite having only one hit (252), this test highlights the flexibility of fully associative mapping, as it can store any block in any cache line. However, the lack of a systematic replacement strategy can lead to suboptimal hit rates, particularly in scenarios with limited spatial locality.

6.3 Testing: Fully Associative Mapping with LRU Replacement Policy

This section evaluates the performance of the cache simulator configured with a fully associative mapping technique and the **Least Recently Used (LRU)** replacement policy. The LRU policy evicts the cache block that has been unused for the longest period.

Test Configuration

- **Memory size**: 256 bytes
- **Cache size**: 32 bytes
- **Block size**: 4 bytes
- **Mapping technique**: Fully Associative
- **Replacement policy**: LRU (Least Recently Used)

The memory address sequence used for this test is: 89, 63, 139, 205, 122, 137, 71, 86, 152, 69

Address Translation and Results

For each address, the cache simulator determines whether the access results in a hit or a miss. If a cache block needs to be evicted due to limited space, the LRU policy is applied to remove the least recently accessed block.

Memory Address: 89 (0x59)

- **Binary Representation**: 01011001
- **Block Number**: 89 // 4 = 22
- Result: **Miss** (Block 22 loaded into the cache).

Memory Address: 63 (0x3F)

- **Binary Representation**: 00111111
- **Block Number**: 63 // 4 = 15
- Result: **Miss** (Block 15 loaded into the cache).

Memory Address: 139 (0x8B)

- **Binary Representation**: 10001011
- **Block Number**: 139 // 4 = 34
- Result: **Miss** (Block 34 loaded into the cache).

Memory Address: 205 (0xCD)

- **Binary Representation**: 11001101
- **Block Number**: 205 // 4 = 51
- Result: **Miss** (Block 51 loaded into the cache).

Memory Address: 122 (0x7A)

- **Binary Representation**: 01111010
- **Block Number**: 122 // 4 = 30
- Result: **Miss** (Block 30 loaded into the cache).

Memory Address: 137 (0x89)

- **Binary Representation**: 10001001
- **Block Number**: 137 // 4 = 34
- Result: **Hit** (Block 34 was recently accessed and is still in the cache).

Memory Address: 71 (0x47)

- **Binary Representation**: 01000111
- **Block Number**: 71 // 4 = 17
- Result: **Miss** (Block 17 loaded into the cache; block 22, which was least recently used, was evicted).

Memory Address: 86 (0x56)

- **Binary Representation**: 01010110
- **Block Number**: 86 // 4 = 21
- Result: **Miss** (Block 21 loaded into the cache).

Memory Address: 152 (0x98)

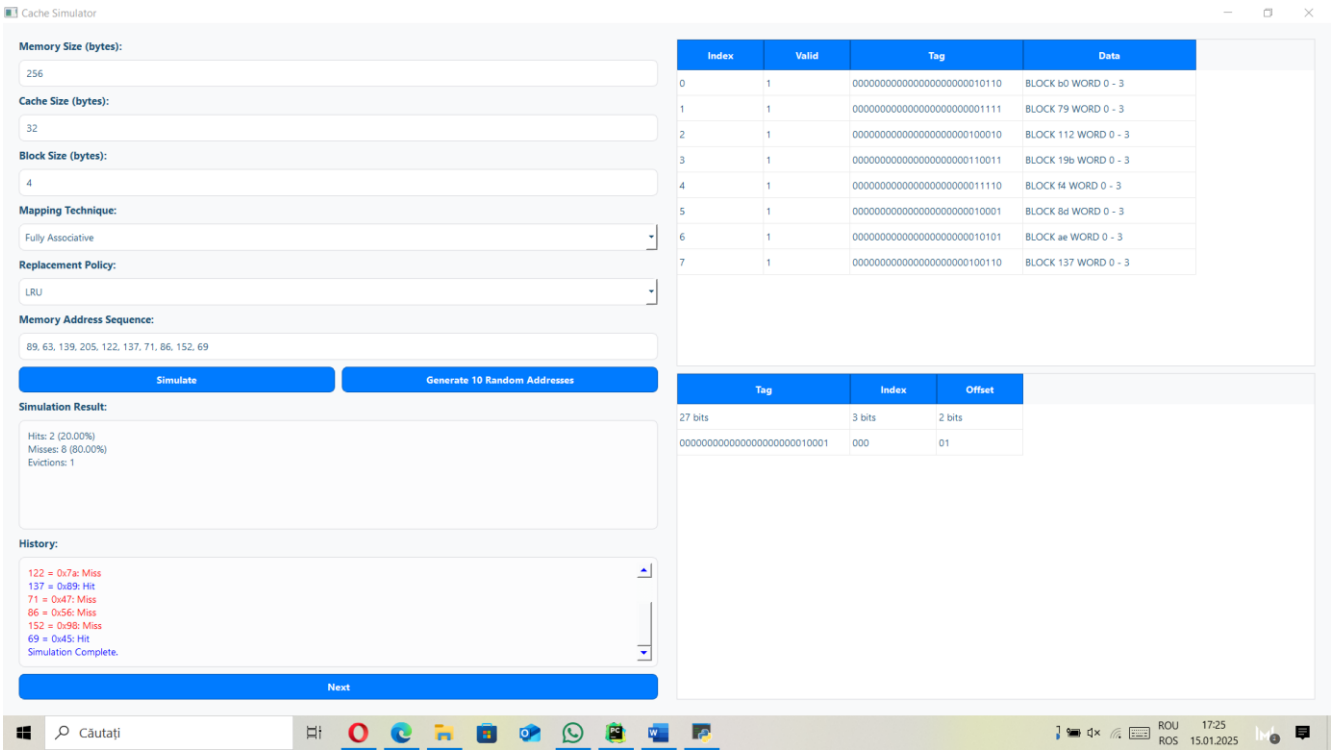
- **Binary Representation**: 10011000
- **Block Number**: 152 // 4 = 38
- Result: **Miss** (Block 38 loaded into the cache).

Memory Address: 69 (0x45)

- **Binary Representation**: 01000101
- **Block Number**: 69 // 4 = 17
- Result: **Hit** (Block 17 was recently accessed and is still in the cache).

Summary of Results

- **Hits**: 2 (20.00%)
  - Addresses 137 and 69.
- **Misses**: 8 (80.00%)
- **Evictions**: 1
  - Block 22 was evicted during the access to address 71.



Analysis

The **LRU replacement policy** effectively maintains blocks that have been recently used, leading to two hits during this simulation. The eviction process ensures that the least recently accessed blocks are removed when the cache reaches capacity. Despite the moderate hit rate of 20%, the LRU policy demonstrates its potential to adapt to patterns of temporal locality, as seen with repeated accesses to 137 and 69.

Conclusions

The Cache Simulator project provided an in-depth exploration of cache memory operations, focusing on two primary mapping techniques: direct mapping and fully associative mapping. The implementation allowed for the simulation of diverse scenarios to analyze the behavior and performance of these configurations under different access patterns.

Implemented Features

In this project, I implemented **direct mapping** and **fully associative mapping** cache configurations. For fully associative mapping, five distinct replacement policies were incorporated: **Least Frequently Used (LFU)**, **Least Recently Used (LRU)**, **First-In-First-Out (FIFO)**, **Most Recently Used (MRU)**, and **Random**. This provided a comprehensive view of how different policies influence cache performance metrics like hit rates, miss rates, and eviction patterns.

Conclusion on Performance Analysis

The performance of cache systems varies significantly based on the mapping technique and the replacement policy employed. Through rigorous testing and simulations, the following insights were derived:

Direct Mapping

Direct mapping is the simplest cache organization technique, where each block of main memory maps to a unique cache line. While it is computationally efficient due to its straightforward implementation, it suffers from **conflict misses** when multiple memory blocks map to the same cache line. This can lead to poor performance in scenarios with non-uniform or repetitive access patterns. In our tests, direct mapping consistently resulted in a higher miss rate due to its rigidity in mapping.

Fully Associative Mapping

Fully associative mapping offers greater flexibility by allowing any memory block to occupy any cache line. This significantly reduces conflict misses compared to direct mapping. However, its performance is heavily influenced by the replacement policy:

1. **Least Recently Used (LRU)**

LRU is efficient in maintaining frequently accessed data in the cache, making it a strong choice for workloads with temporal locality. Our tests showed that LRU achieved a higher hit rate compared to most other policies, especially in access patterns where recently used data was repeatedly accessed.

2. **Least Frequently Used (LFU)**

LFU keeps blocks with the highest access frequency in the cache. While effective in certain workloads with highly repetitive patterns, LFU can struggle with dynamic or changing access patterns, where it may retain blocks that are no longer needed, leading to suboptimal performance.

3. **First-In-First-Out (FIFO)**

FIFO evicts the oldest block regardless of its usage frequency or recency. This policy is simple to implement but often underperforms compared to LRU and LFU, especially in scenarios with localized access patterns, as it may evict useful data prematurely.

4. **Most Recently Used (MRU)**

MRU, which evicts the most recently accessed block, is suitable for specific workloads where older data is more likely to be reused. However, in general-purpose scenarios, its performance is inconsistent, often resulting in a lower hit rate compared to LRU or Random.

5. **Random**

The random replacement policy evicts a cache line arbitrarily. While not optimal for maximizing performance, it avoids the overhead of maintaining access patterns and can outperform structured policies like FIFO in scenarios with unpredictable access patterns.

Efficiency Observations

Direct Mapping: Efficient in hardware and suitable for simple systems but prone to high miss rates.

- **Fully Associative with LRU**: Offers the best balance of hit rate and complexity, making it ideal for systems with temporal locality.
- **Fully Associative with LFU**: Performs well in scenarios with static, repetitive access patterns but struggles with dynamic workloads.
- **FIFO and Random**: Provide acceptable performance for general-purpose workloads but lack the optimization seen in LRU or LFU.
- **MRU**: Niche use cases where recent data is less critical can benefit from MRU, but its performance is not consistent across different scenarios.

Overall Conclusion

The choice of cache organization and replacement policy significantly impacts system performance. Direct mapping is efficient for hardware simplicity but limited by conflict misses. Fully associative mapping, while more complex, provides better performance when paired with adaptive replacement policies like LRU or LFU. For optimal results, the selection of a cache configuration should be tailored to the specific workload and access patterns of the system.

Bibliography

[1] D. A. Patterson and J. L. Hennessy, "**Computer Organization and Design: the Hardware/ Software Interface**" 5th edition, ed. Morgan–Kaufmann, 2014

[2] D. A. Patterson and J. L. Hennessy, "**Computer Organization and Design: A Quantitative Approach**" 5th edition, ed. Morgan–Kaufmann, 2011

[3] GeeksforGeeks, **"Cache Memory"**
https://www.geeksforgeeks.org/cache-memory/

[4] Carnegie Mellon University, **"Cache Memory and Performance"**
https://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/lectures/cache.pdf

[5] TutorialsPoint, **"Cache Memory in Computer Architecture"**
https://www.tutorialspoint.com/cache-memory-in-computer-organization

[6] Computer Science Bytes, **"Cache Memory – Write Policies"**
https://computerbytes.org/cache-memory-write-policies

[7] University of Washington, **"Memory Hierarchy and Cache Memory"**
https://courses.cs.washington.edu/courses/cse351/19sp/slides/12-cache-memory.pdf