

# **Cache Memory Simulator**

Student: Pop Alexandra

---

Structure of Computer Systems Project

---

Technical University of Cluj-Napoca

January 15, 2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context .....	1
1.2	Objectives .....	1
<b>2</b>	<b>Bibliographic Research</b>	<b>1</b>
2.1	What is a cache memory? .....	1
2.2	Challenges in Cache Memory Design. ....	3
2.3	How the Initial Part of the Project Will Be Developed. ....	5
<b>3</b>	<b>Analysis</b>	<b>6</b>
3.1	Project Proposal .....	6
3.2	Project Analysis .....	6
	3.2.1 Architecture Flexibility .....	7
	3.2.2 Multi-Level Caching .....	8
	3.2.3 Impact of Replacement Policies .....	9
	3.2.4 Performance Metrics and Analysis .....	10
	3.2.5 Real-Time Visualization of Cache Behavior .....	10
	3.2.6 Simulating Access Patterns to Show Spatial and Temporal Locality .....	10
<b>4</b>	<b>Design</b>	<b>11</b>
4.1	User Interface Layout .....	12
4.2.	Core Functional Design .....	14
4.3.	Data Flow and Operational Logic .....	16
4.4.	Planned Enhancements .....	18
<b>5</b>	<b>Implementation</b>	<b>19</b>

5.1 Core Flask Application and API Routing: app.py . . . . .	19
5.2 Cache Logic and Core Functionality: cache_simulator.py . . . . .	22
5.3 Frontend Logic and API Interaction: script.js . . . . .	25
5.4 User Interface Structure: index.html . . . . .	26
5.5 User Interface Styling: style.css . . . . .	28
5.6 User Manual . . . . .	31
5.6.1 Configuration Section . . . . .	31
5.6.2 Manual Access Section . . . . .	32
5.6.3 Batch Operations Section . . . . .	32
5.6.4 Cache Statistics Section . . . . .	33
5.6.5 Cache Contents (VDT Cache Data) . . . . .	33
5.6.6 Physical Memory . . . . .	34
5.6.7 Performance Metrics and Graphs. . . . .	34
5.6.6 History . . . . .	35
6. Summary . . . . .	35
7. Conclusion . . . . .	
<b>Bibliography</b>	<b>36</b>

# Introduction

## 1.1 Context

In modern computer systems, cache memory plays a critical role in bridging the speed gap between the CPU and the main memory. Cache memory acts as a temporary storage area for frequently accessed data, allowing for faster data retrieval and reduced latency. It uses principles of locality—spatial and temporal—to predict and store the most relevant data, improving overall system performance.

This project involves creating a cache memory simulator designed to model various cache configurations and policies. By simulating how a cache interacts with the CPU and main memory, this tool can help analyze and optimize memory performance for different scenarios.

## 1.2 Objectives

The objective of this project is to design a flexible cache memory simulator capable of:

- To model various cache architectures, including direct-mapped, set-associative, and fully associative caches.
- To simulate multiple replacement policies, such as Least Recently Used (LRU), FIFO, and random replacement.
- To evaluate performance metrics like cache hit/miss rates and average memory access time.

*List 1.2.: The objective of this project*

Additionally, the simulator will provide a visualization of cache usage and performance data, allowing for better analysis of how various configurations and access patterns affect cache behavior.

# Bibliographic Research

## 2.1 What is a Cache Memory?

Cache memory is a high-speed memory layer designed to reduce the time the CPU takes to retrieve frequently accessed data, thus enhancing overall system performance. By storing copies of commonly used data closer to the CPU, cache memory helps reducing the latency caused by slower main memory (RAM), which can otherwise become a bottleneck.

In simple terms, every memory access is first examined for the presence in the Cache. Figure 2.1.1 envisions the access path between CPU, Cache and Main Memory.

When the CPU initiates READ on a memory location,

- It is checked and determined whether it is available in the Cache
- If available, at the access rate of Cache, the data is returned to CPU
- If not available, access is made to main memory. The particular word is supplied to CPU and also written in the Cacheline of the Cache. (This is required keeping in line with the Principle of Locality of Reference)

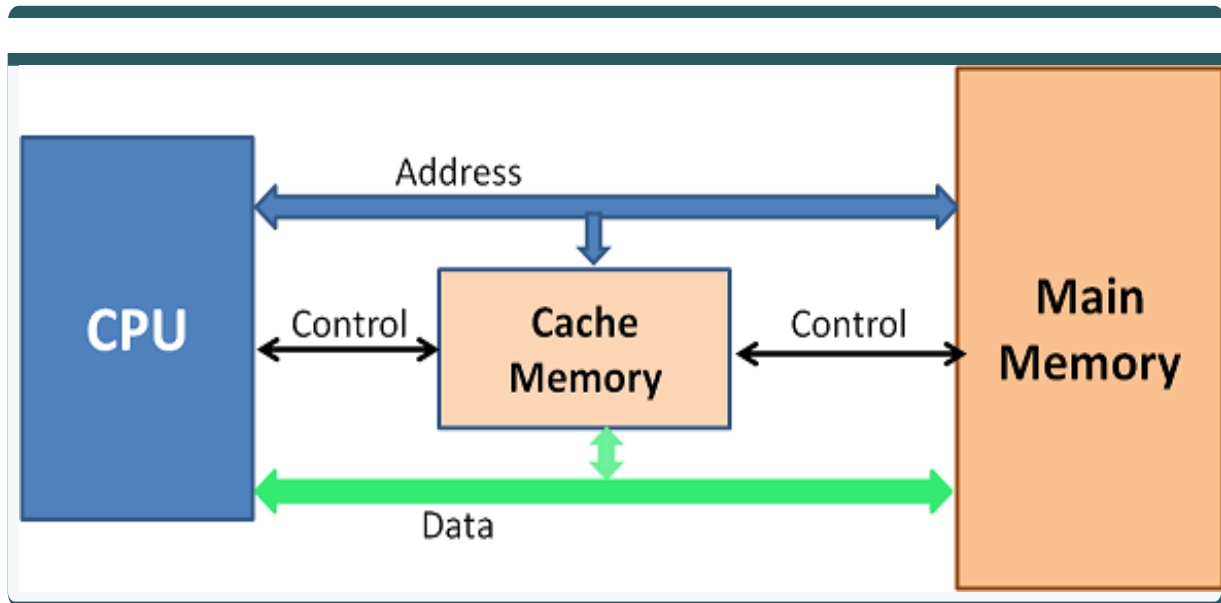


Figure 2.1.1: Structure of Cache Memory

Cache design is primarily guided by two principles of locality:

- **Spatial Locality:** When a particular memory location is accessed, nearby locations are likely to be accessed soon.
- **Temporal Locality:** When a specific memory location is accessed, it is likely to be accessed again in the near future.

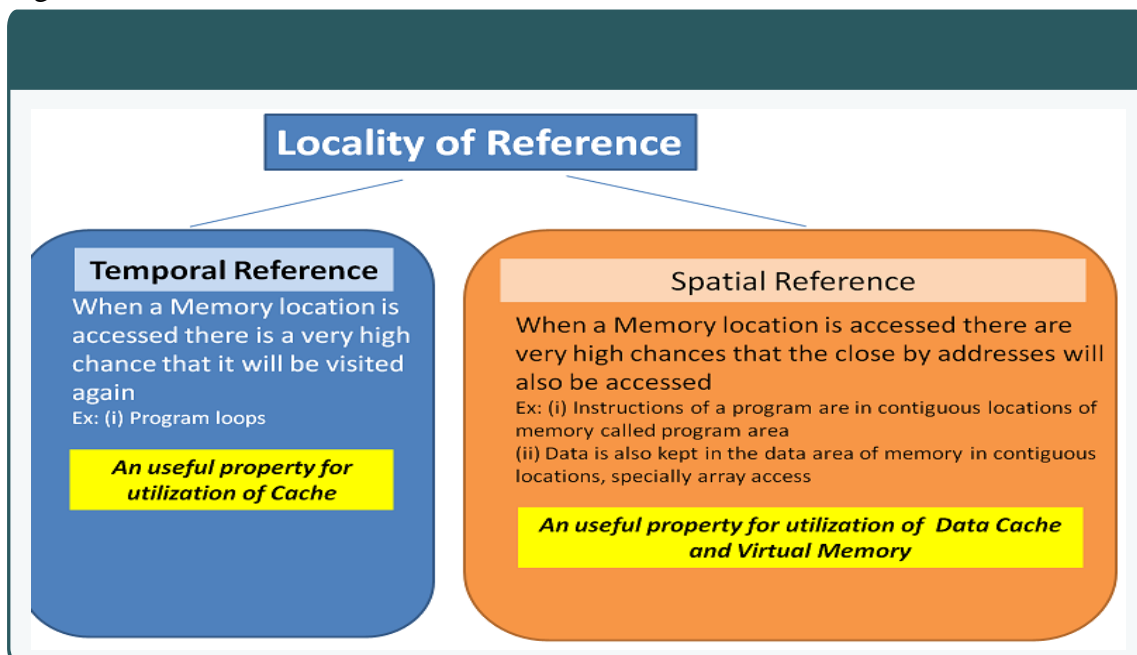


Figure 2.1.2: Locality of Reference

Caches are typically organized by levels (L1, L2, and L3), based on their proximity to the CPU, each providing different trade-offs in speed and capacity. Depending on the architecture, caches can also be implemented in various ways:

- **Direct-Mapped Cache:** Each memory location maps to exactly one cache line, offering simplicity but increasing the chance of conflict misses.
- **Set-Associative Cache:** Each memory block can be mapped to any line within a specific set, balancing flexibility and complexity.
- **Fully Associative Cache:** Any memory block can be stored in any cache line, minimizing conflicts but increasing search time and complexity.

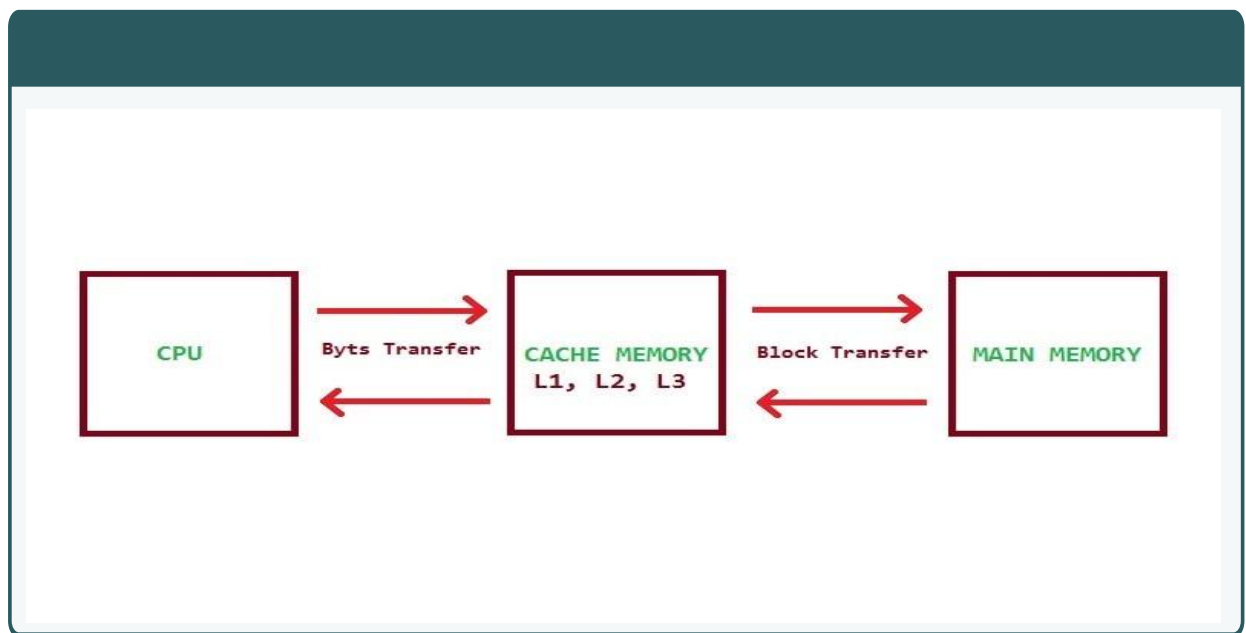


Figure 2.1.3: Cache Architectures

In this simulator, the objective is to model these cache types to explore how they impact performance metrics such as cache hit rates and miss rates. By simulating and visualizing these different cache implementations, the project demonstrates how cache memory plays a critical role in reducing CPU access times and improving system efficiency.

## 2.2 Challenges in Cache Memory Design

Designing an efficient cache memory system involves balancing speed, complexity, and cost. Given its limited size, cache memory can only store a subset of data from the main memory, leading to several design challenges:

### 1. Fully Associative Cache Support (L3 Cache)

- **Feature:** I need to add a fully associative option where any block of data can be placed in any line of the cache, allowing maximum flexibility.
- **Implementation:** Configuration of the cache to have a single set with multiple blocks, making every block accessible for any data.

- **Usefulness:** This modification would complete cache architecture options, allowing users to compare direct-mapped, set-associative, and fully associative caches, each with unique trade-offs.

## 2. Performance Metrics

- **Hit and Miss Rates:**
  - Calculate the percentage of hits and misses rather than just their counts.
  - Display these values as:

$$\text{Hit Rate} = \frac{\text{Hits}}{\text{Total Accesses}} \times 100\%$$

$$\text{Miss Rate} = \frac{\text{Misses}}{\text{Total Accesses}} \times 100\%$$

- **Average Memory Access Time (AMAT):**
  - Calculate AMAT based on hit time, miss rate, and miss penalty:

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

- **Implementation:** Define hit time (time to access cache) and miss penalty (time to access main memory) as constants or configurable parameters, then calculate and display AMAT.
- **Usefulness:** These metrics are essential for assessing cache efficiency, allowing users to understand the practical impact of different cache configurations.

## 3. Enhanced Visualization and Feedback

- **Real-Time Graphs:** Add real-time line charts or bar graphs to display hit and miss rates over time or access operations, helping users visually track cache performance trends.
- **Cache Eviction Details:** Provide detailed feedback on which data blocks are evicted and why (e.g., due to LRU or FIFO policy), which would help users understand replacement policies better.

## 4. Simulation of Spatial and Temporal Locality

- **Feature:** Simulate access patterns that emphasize spatial or temporal locality.
- **Implementation:** Introducing sample access sequences that the user can choose from, such as sequential (emphasizing spatial locality) or repeated accesses to the same address (emphasizing temporal locality).
- **Usefulness:** Demonstrating locality patterns would help users understand the foundational principles of cache design and how caches optimize access time.

## 2.3 How the Initial Part of the Project Will Be Developed

The cache memory simulator will be developed in stages, starting from basic operations and gradually incorporating more sophisticated features and configurations. The initial phases are outlined below:

### Step 1: Simulator Framework and Data Structures

- **Core Simulator Structure:** The simulator's foundation involves setting up essential data structures to model the cache. Each cache line (or block) will hold key elements like the data, tag, and metadata for tracking usage and replacement (e.g., last access time for LRU).
- **Data Structures in JavaScript and Python:** In JavaScript (frontend) and Python (backend), arrays and objects will represent cache blocks and sets. Each block contains fields for validity, dirtiness, the data itself, and metadata for replacement policies (like last access time for LRU and load order for FIFO).

### Step 2: Implementing Cache Mapping Techniques

- **Direct-Mapped Cache:** The simulator's first configuration will be a direct-mapped cache. Each memory address will map to one specific cache line, determined by splitting the address into tag, index, and offset. Functions will extract these parts from each memory address, allowing the simulator to check if a requested address is in cache (hit) or must be loaded from main memory (miss).
- **Set-Associative and Fully Associative Cache:** Expanding from direct-mapped, the simulator will allow set-associative and fully associative caches. Set-associative mapping divides the cache into sets, with each set containing multiple blocks (or ways), while fully associative allows any block to be placed anywhere in the cache.

### Step 3: Implementing Cache Replacement Policies

- **Least Recently Used (LRU):** In LRU, each cache block stores the time of its last access. The simulator will replace the least recently used block within a set when space is needed.
- **First-In-First-Out (FIFO):** FIFO replaces the oldest block in the cache. The simulator will keep a load order for each block, removing the block that was loaded first within a set.
- **Random Replacement:** This simple policy randomly selects a block within a set for replacement when new data needs to be loaded.

These replacement policies are selectable in the simulator, allowing users to test each one and observe its impact on cache performance.



# Analysis

## 3.1 Project Proposal

The simulator is implemented using Python (backend) and JavaScript (frontend), ensuring efficient modeling of memory operations, data visualization, and performance tracking. It includes modules to handle memory access, cache organization, and replacement policies, each designed to simulate real-world cache behavior.

- Configurable Cache Architectures: Direct-mapped, set-associative, and fully associative caches.
- Performance Metrics: Real-time tracking of hits, misses, hit rate, miss rate, and AMAT.
- Replacement Policies: Options for LRU, FIFO, and Random replacement strategies.
- Real-Time Visualization: Visuals for cache contents, hits, misses, evictions, and performance graphs.
- Access Pattern Simulation: Simulated access patterns for spatial and temporal locality.

*List 3.1: Features of the final Cache Memory*

## 3.2 Project Analysis

The Cache Memory Simulator is designed to provide a comprehensive platform for understanding the complexities of cache memory in computer systems. By allowing users to configure various cache parameters, observe performance metrics, and visualize cache behavior in real-time, the simulator addresses the critical aspects of cache design and performance analysis.

### 3.2.1 Architecture Flexibility

The simulator currently supports **direct-mapped** and **set-associative caches**, with plans to add **fully associative caching** to complete the range of possible cache architectures. Each architecture type handles data placement and conflict management differently, and the simulator allows users to experiment with these setups to see how associativity affects performance:

- **Direct-Mapped Cache:** This architecture restricts each memory block to a specific cache line, leading to fast access but a higher likelihood of **conflict misses** when multiple addresses map to the same cache line.



Figure 3.21: Direct-Mapped Cache

- Set-Associative Cache:** This configuration allows a set number of blocks per set (e.g., 2-way or 4-way), offering a balance between the simplicity of direct-mapped caches and the flexibility of fully associative caches. Users can observe reduced conflict misses with increased associativity.
- Future Fully Associative Cache:** In a fully associative setup, any data block can go in any cache line, providing maximum flexibility and minimizing conflicts but increasing search complexity. Implementing this architecture will show users the ultimate trade-off between flexibility and access time.

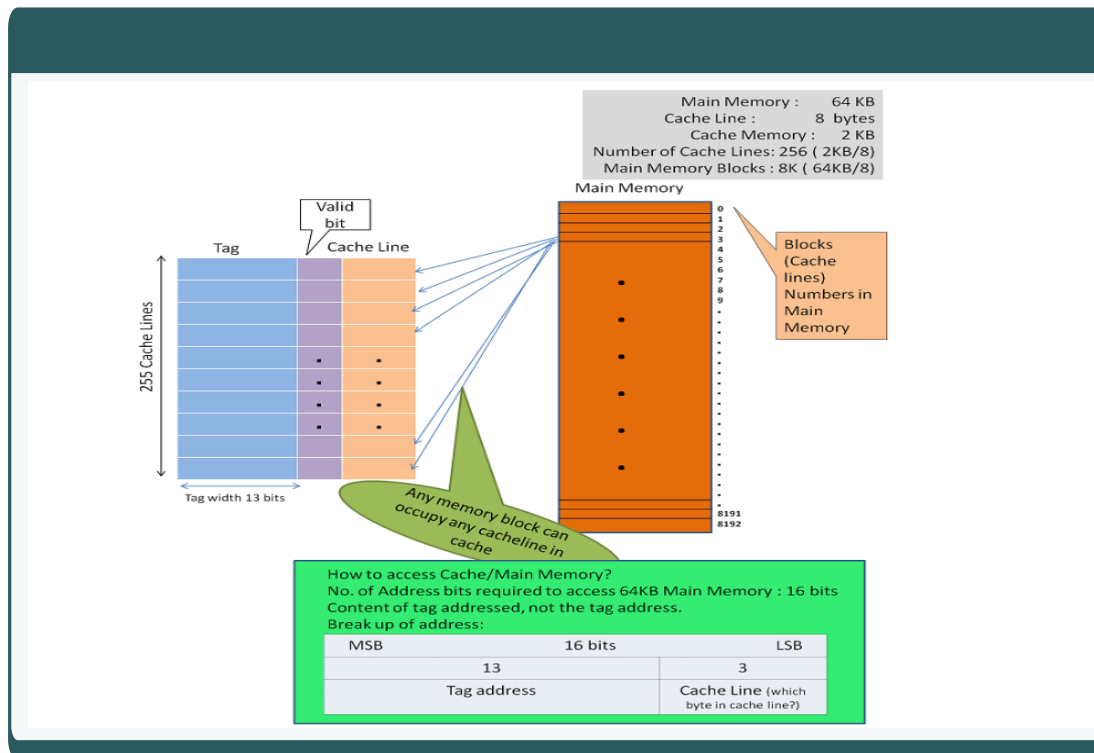


Figure: Multi-Mapped Cache

### 3.2.2 Multi-Level Caching

The simulator will include multi-level caching to mimic the structure of real-world systems, where caches are organized into **L1, L2, and L3 levels** with distinct sizes and speeds. This structure illustrates how layered caches work together to optimize access times:

- **L1 Cache** (small and fast): Located closest to the CPU, this cache level provides immediate access to frequently accessed data but is limited in size.
- **L2 Cache** (larger and slower): Positioned between L1 and main memory, it offers more storage for frequently accessed data with a slightly longer access time.
- **L3 Cache** (largest and slowest): Often fully associative, L3 provides a final caching layer that minimizes access to main memory.

Each level will be configurable, allowing users to experiment with various sizes and speeds to see how multi-level caching reduces average access times by retaining the most critical data close to the CPU. By adjusting parameters for each level, users will gain insights into the benefits of cache hierarchy in balancing speed, storage, and access frequency.

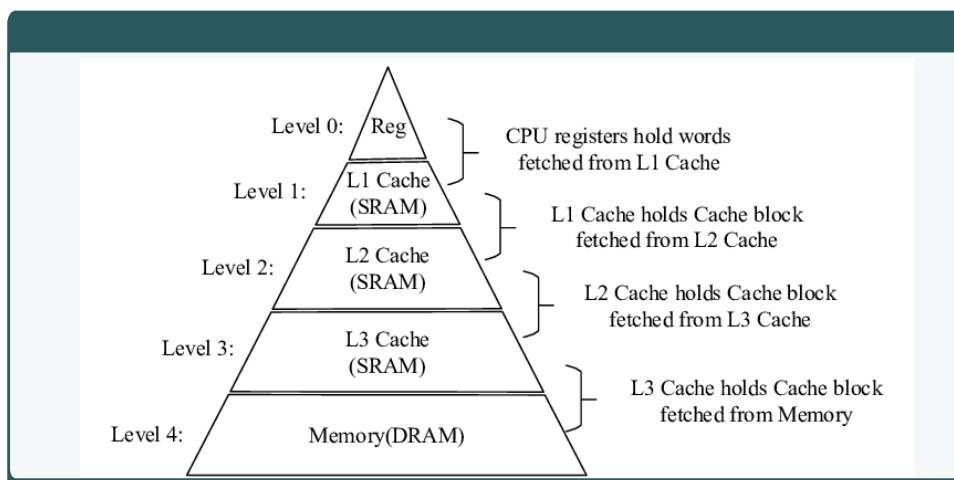


Figure: Cache Levels

### 3.2.3 Impact of Replacement Policies

Users can configure the cache's **replacement policy**, which determines which data is evicted when the cache is full. The simulator currently supports **Least Recently Used (LRU)**, **First-In-First-Out (FIFO)**, and **Random Replacement** policies, each offering unique strategies for managing cache data:

- **LRU**: Retains recently accessed data for as long as possible, leveraging **temporal locality** by prioritizing frequently accessed data. This policy is ideal for situations where data is repeatedly accessed within short time intervals.

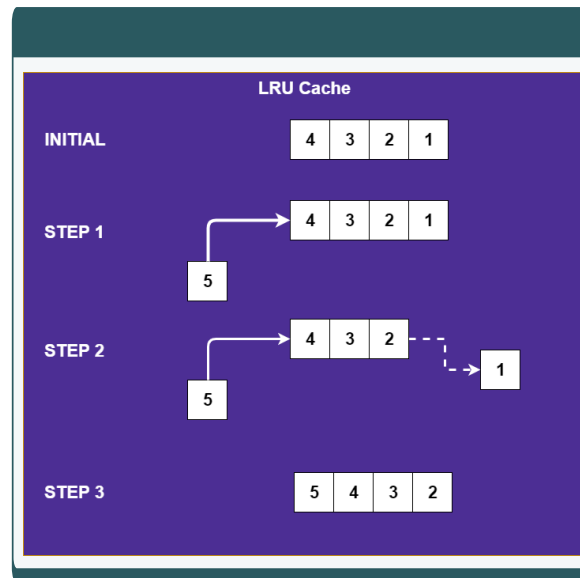


Figure: Least Recently Used Replacement

- **FIFO**: Evicts the oldest data in the cache first, a simpler policy that does not consider data access frequency but ensures that data doesn't remain in the cache indefinitely.

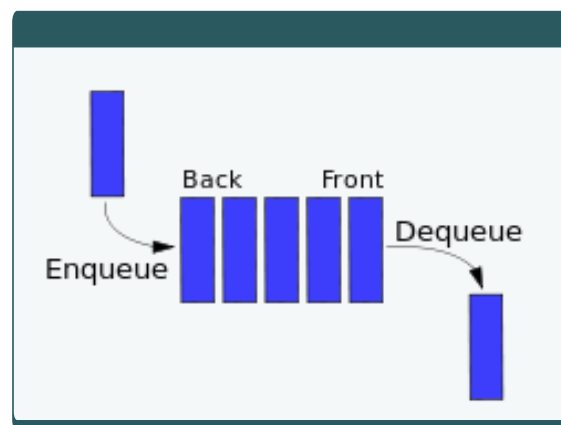


Figure: First-In-First-Out Replacement

- **Random Replacement**: Selects data to evict randomly, providing a baseline approach that can sometimes reduce conflicts without complex tracking.

By experimenting with these policies, users can observe how different strategies impact cache performance and understand why certain replacement policies are preferred in specific architectures.

## 3.2.4 Performance Metrics and Analysis

The simulator currently tracks basic performance metrics, specifically **cache hits** and **misses**, to provide users with immediate feedback on how well their configurations are working. Future updates will add more detailed metrics:

- **Hit and Miss Rates:** Calculated as percentages, these rates offer a normalized measure of cache efficiency, showing users how often their cache configuration successfully retrieves data without accessing main memory.
- **Average Memory Access Time (AMAT):** AMAT will provide a clear picture of the overall access speed improvement from caching, calculated by combining hit time, miss rate, and miss penalty. By configuring access times, users can see how each cache level affects AMAT, offering a quantifiable view of the trade-offs between speed and storage.

These metrics will enable users to perform detailed performance analysis and make data-driven decisions about their cache configurations, helping them see the practical impact of their design choices.

## 3.2.5 Real-Time Visualization of Cache Behavior

The simulator's interface includes visual representations of cache contents, with **color-coded highlights for hits and misses**, which are particularly valuable for understanding cache behavior. As users perform read and write operations, the interface highlights which addresses are found in the cache (hits) and which are not (misses). This feature will be enhanced with:

- **Graphical Representations of Hit/Miss Trends:** Real-time graphs will track hit and miss rates over time, allowing users to visually observe how changes in configuration affect performance trends.
- **Detailed Eviction Information:** Users will be able to see which data blocks are evicted and why (based on the replacement policy), helping them understand how each replacement strategy impacts cache contents.

This immediate visual feedback helps demystify cache behavior and shows users how data moves through the cache in response to their settings.

## 3.2.6 Simulating Access Patterns to Show Spatial and Temporal Locality

Cache design relies on **spatial locality** (data near recently accessed locations will likely be accessed soon) and **temporal locality** (recently accessed data will likely be accessed again). The simulator will allow users to test these principles by simulating various access patterns:

- **Sequential Access (Spatial Locality):** Simulates accessing data in consecutive memory locations, helping users see how caches perform when nearby data is frequently accessed.
- **Repeated Access (Temporal Locality):** Models repeated accesses to the same memory location, demonstrating how the cache retains frequently used data for quick access.

By observing how different configurations handle these patterns, users can understand the core principles that drive cache performance improvements and see how cache structure adapts to common access behaviors.

## Design

The **Cache Memory Simulator** is designed to provide an interactive experience that enables users to configure, visualize, and analyze the workings of a cache memory system. With features for manual cache access, real-time statistics, and configuration options, the simulator offers a comprehensive tool for exploring cache concepts in computer architecture.

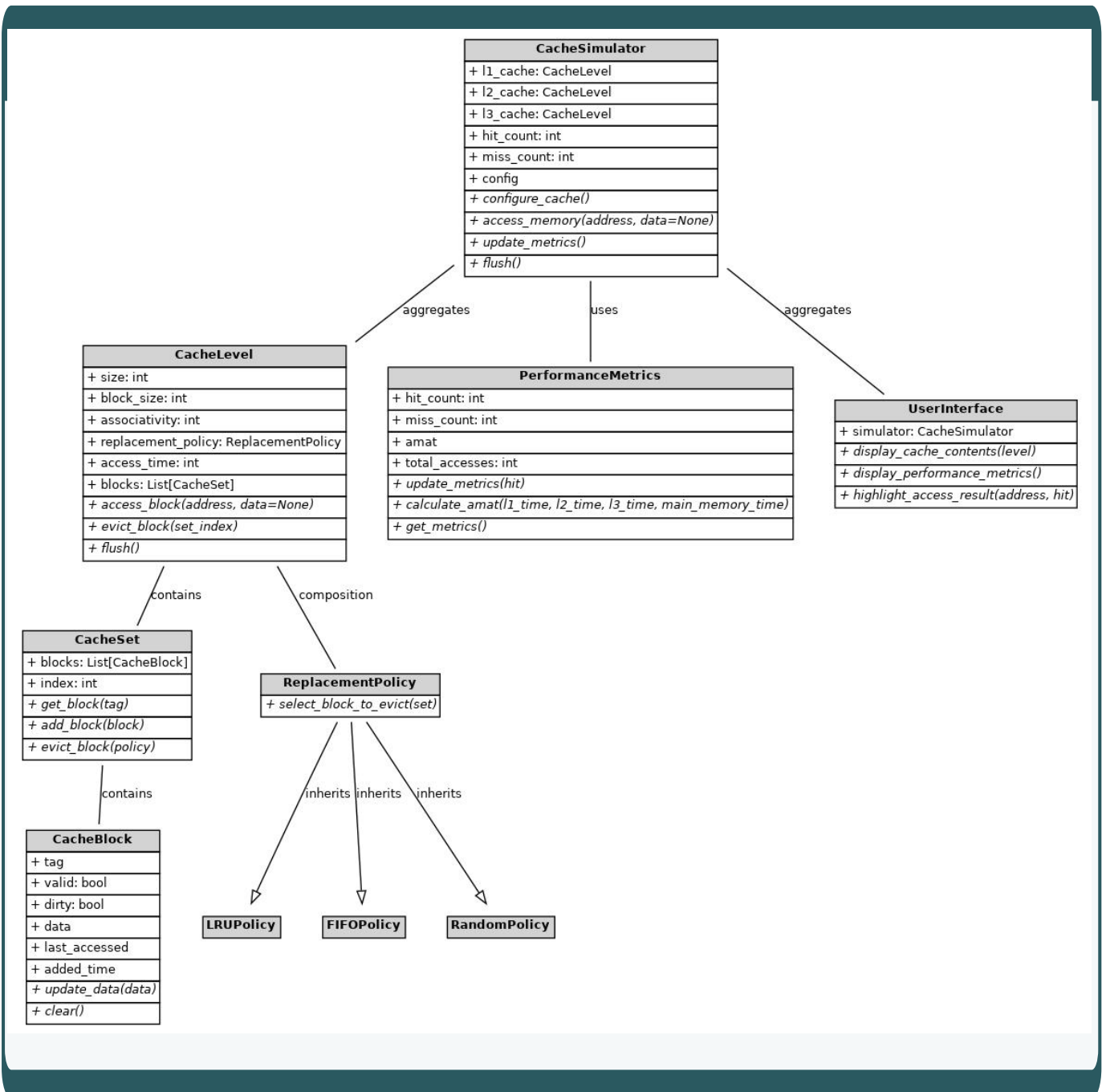


Figure: Class Diagram

## 4.1. User Interface Layout

The user interface (UI) is structured to be intuitive and clearly segmented into functional sections, making it easy for users to navigate and experiment with different configurations. Here's a breakdown of each section in the UI:

- **Configuration Section:**
  - This section allows users to set essential parameters for the cache, including:
    - **Address Width:** Number of bits used for memory addresses.
    - **Cache Size:** Total size of the cache in bytes.
    - **Block Size:** Size of each cache block.
    - **Associativity:** Determines the cache type, e.g., 1-way for direct-mapped or higher for set-associative.
    - **Replacement Policy:** Allows selection of the eviction strategy, such as FIFO, LRU, or Random.
  - After setting these options, users click “Generate System” to initialize the cache based on their configuration.
- **Manual Access Section:**
  - This section enables users to manually interact with the cache by entering a memory address and optional data.
  - **Read** and **Write** buttons let users simulate read or write operations on specific addresses, allowing them to see how the cache responds.
  - **Flush** clears all cached data, resetting the system to observe cache behavior from a fresh state.
- **Batch Processing Section**
  - Users can perform batch operations to simulate multiple read and write requests at once.
  - **Input Format:**
    - Each operation is entered on a separate line (e.g., READ 0xA, WRITE 0xB 0x5F).
  - **Features:**
    - The **Run Batch** button processes the input operations sequentially, providing real-time results for each.
    - Results are displayed below the input field, showing the outcome (e.g., hit, miss) for each operation.
    - Batch results are also added to the **History** section for a chronological record of operations.
- **Cache Statistics**
  - Displays real-time **performance metrics**, including:
    - **Cache Hits:** Number of successful data retrievals from the cache.
    - **Cache Misses:** Number of times the requested data was not found in the cache.
    - **Average Memory Access Time (AMAT) (New):** Dynamically calculated using the number of hits, misses, cache access time, and main memory access time.
  - Helps users analyze and compare the efficiency of various cache configurations.
- **Cache Contents (VDT Cache Data)**
  - A structured table displays the current state of the cache, including:
    - **Columns** for each byte in a cache block.
    - Metadata columns:
      - **V (Valid Bit):** Indicates if the cache line contains valid data.

- **D (Dirty Bit):** Indicates if the data in the cache block differs from main memory.
  - **T (Tag):** Stores the tag value to identify data blocks.
- Users can observe which data is currently cached and monitor changes during operations.
- **Physical Memory**
  - Displays the main memory's current state with:
    - **Addresses.**
    - **Corresponding byte values.**
  - Provides insights into how the cache interacts with the main memory, helping users understand the optimization process.
- **Performance Metrics Graph (New)**
  - Visualizes cache performance over time using a **line chart** that dynamically updates with each operation:
    - **Cache Hits:** Represented in green.
    - **Cache Misses:** Represented in red.
    - **AMAT:** Represented in blue.
  - The graph provides a visual representation of the cache's efficiency, enabling users to identify trends and performance bottlenecks.
- **History**
  - Tracks all cache-related operations (manual and batch) and displays:
    - Operation type (e.g., read, write).
    - Address.
    - Data (if applicable).
    - Result (e.g., hit, miss).
  - Provides a chronological view of cache events, making it easier to analyze usage patterns and the effectiveness of different configurations.

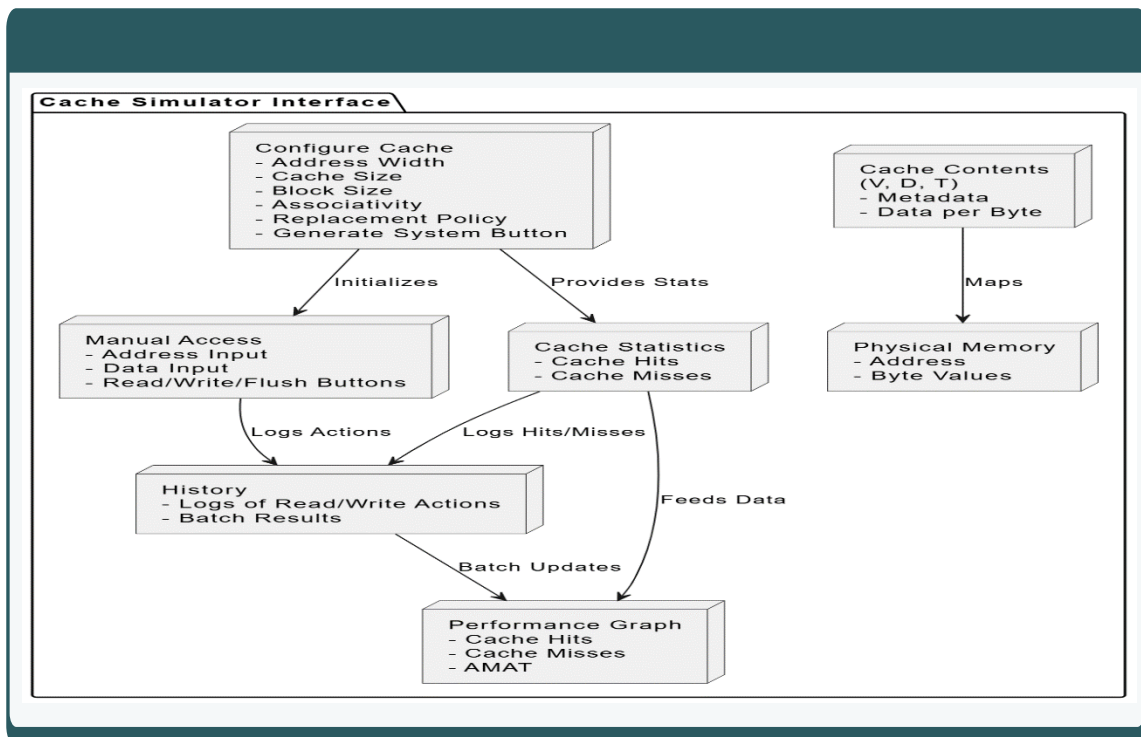


Figure: Cache Simulator Interface



## 4.2. Core Functional Design

Each component of the simulator has a specific function to ensure modularity, maintainability, and flexibility for future extensions:

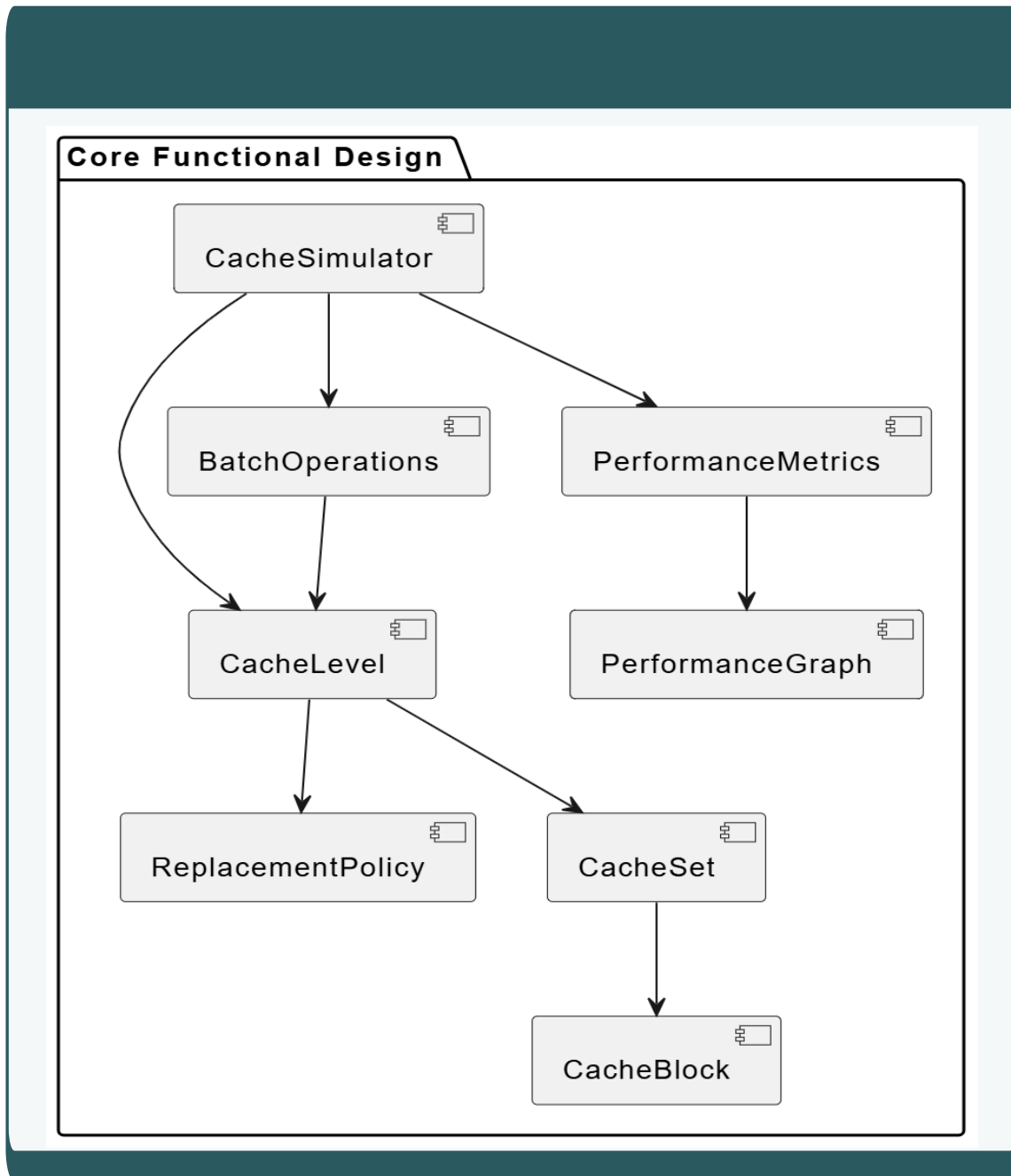


Figure: Core Functional Design

- **CacheSimulator Class:**
  - Central to the simulation, **CacheSimulator** manages configurations, directs cache accesses, and updates metrics.

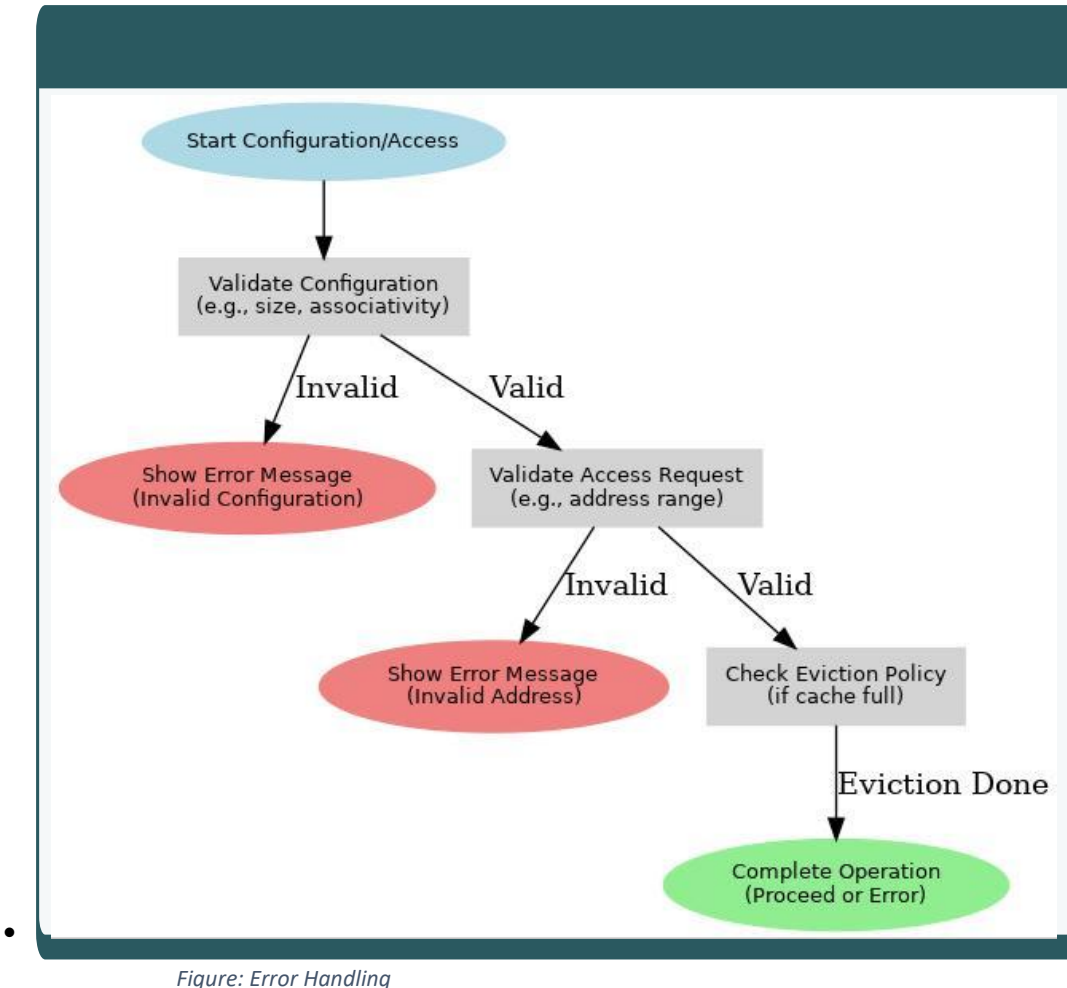
- **ReplacementPolicy Classes:**
  - Implemented as an interface with subclasses (LRUPolicy, FIFOPolicy, RandomPolicy) that define different eviction strategies.
  - This modular approach allows users to swap out policies without altering the CacheLevel class's internal structure.
- **CacheSet and CacheBlock Classes:**
  - CacheSet contains multiple CacheBlock instances, based on the associativity of the cache. This structure allows set-associative caching, where each set can hold several blocks.
  - Each CacheBlock holds metadata (valid, dirty, and tag bits) and data for one memory block, supporting the visualization of cache contents in the UI.
- **PerformanceMetrics Class:**
  - Tracks key performance metrics (hit count, miss count, AMAT) and updates them after each access.
  - This class helps display real-time performance metrics in the UI, showing the impact of user configuration.
- **Batch Operations Support:**
  - Extends the CacheSimulator to process multiple sequential operations in a single request.
  - Tracks individual operation results, including hits, misses, and replacement decisions.
  - Logs batch operation results into the History section for user review.
- **Performance Graph:**
  - Integrated with PerformanceMetrics, the graph dynamically displays:
    - Cache Hits
    - Cache Misses
    - AMAT
  - Provides visual feedback on cache behavior as configurations or workloads change.

## Error Handling and Edge Cases

The simulator includes various mechanisms to handle errors and edge cases gracefully:

- **Configuration Errors:**
  - Parameters like cache size, block size, associativity, and address width are validated to ensure they match supported values. If a parameter is invalid, an error message is shown, and the cache isn't initialized.
- **Access Errors:**
  - Invalid addresses or data (e.g., out-of-range or non-numeric values) trigger a clear error message in the UI, preventing the simulator from crashing.
- **Eviction Edge Cases:**
  - If a cache set is full, CacheLevel calls the ReplacementPolicy to select an eviction candidate. This modular design allows different policies to handle full sets without additional code in CacheLevel.
- **Future-Proofing:**

- By encapsulating error handling within specific classes (like CacheSimulator for configuration errors and CacheLevel for access errors), the design is robust against future feature additions, such as new cache levels or policies.



### 4.3. Data Flow and Operational Logic

The simulator follows a streamlined process to handle each access request:

- 1. Configuration:**
  - Upon clicking “Generate System,” CacheSimulator configures each CacheLevel with parameters chosen in the UI.
  - This setup enables multi-level cache functionality, initializing levels independently to work together in a hierarchical manner.
- 2. Access Requests:**
  - For each access (read or write), CacheSimulator attempts to retrieve the data from the L1 cache first, following a cascade approach if there is a miss.
  - Cache hits are processed within the corresponding level, while misses are escalated to the next level until either a hit occurs or main memory is accessed.
  - **Future Development:** With fully associative caching at L3, CacheSimulator will include more complex search and eviction logic in this final level.
- 3. Eviction Logic:**

- If a cache set is full and an eviction is needed, CacheLevel invokes the selected ReplacementPolicy to determine which block to evict.
  - This modularity ensures that different policies can be applied and tested with minimal structural change to the cache level's main logic.
4. **Performance Metrics Update:**
- Each read or write operation updates the PerformanceMetrics class, increasing hit or miss counters as appropriate.
  - This immediate feedback supports the real-time display in the UI's **Cache Statistics** section, allowing users to see how different configurations impact cache performance.
5. **User Feedback and Visualization:**
- After each access, the UI updates the **Cache Contents** and **Physical Memory** tables to reflect any changes.
  - The **History** section logs each action, showing users the progression of cache events.

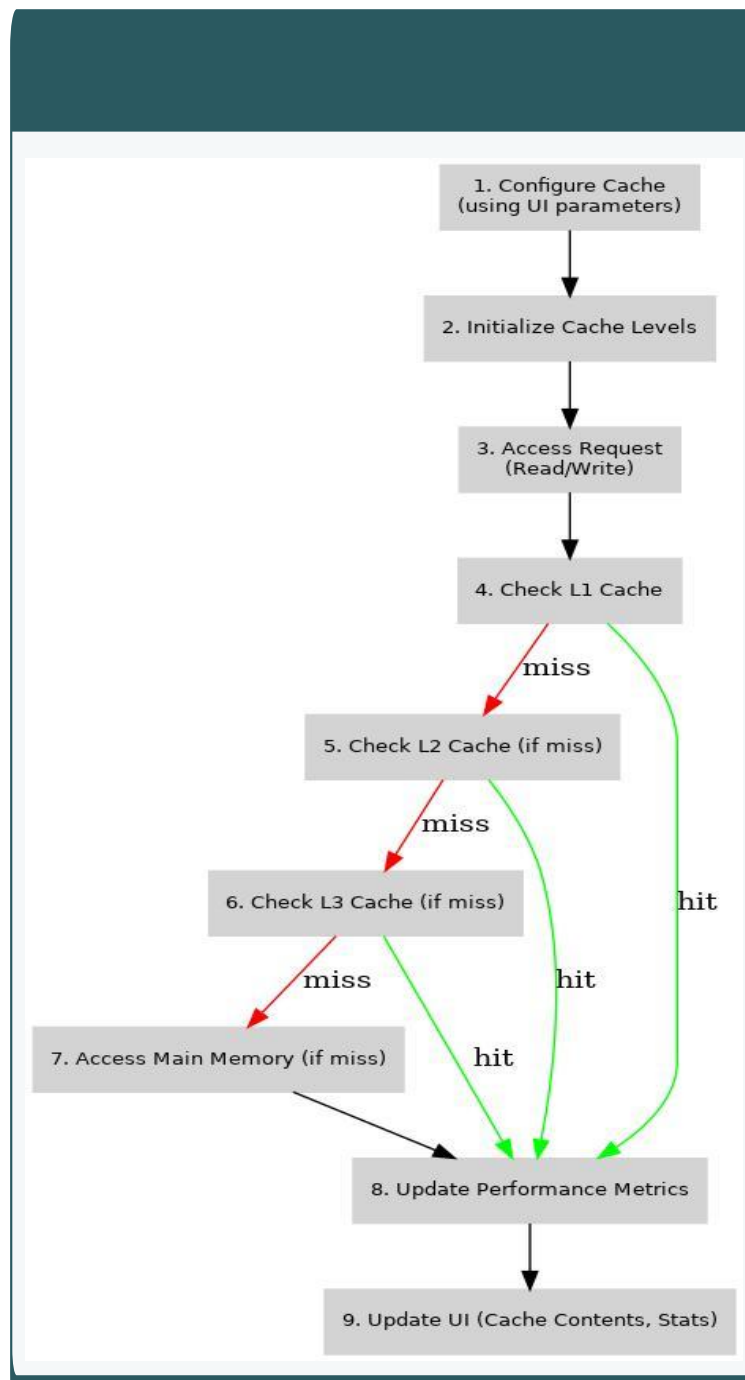


Figure: Data Flow and Operational Logic

# Testing Strategy

To ensure robustness, the simulator includes various testing strategies:

- **Unit Testing:**
  - Each class (e.g., CacheSimulator, CacheLevel, ReplacementPolicy) is tested in isolation to confirm individual methods function as expected.
  - Tests cover core methods, such as `access_block()` in CacheLevel, which checks for hits/misses and manages block access.
- **Integration Testing:**
  - Testing interactions between classes, such as verifying that CacheSimulator correctly manages L1, L2, and L3 caches.
  - Integration tests also confirm that ReplacementPolicy works seamlessly with CacheLevel.
- **UI Testing:**
  - Manual testing of the UI ensures that user inputs trigger the correct backend processes and provide appropriate feedback.
- **Performance Testing:**
  - Performance metrics, such as hit/miss counts and AMAT, are validated under different configurations to ensure accurate tracking and calculations.
  - Tests simulate access patterns to check that performance metrics adjust according to cache usage.

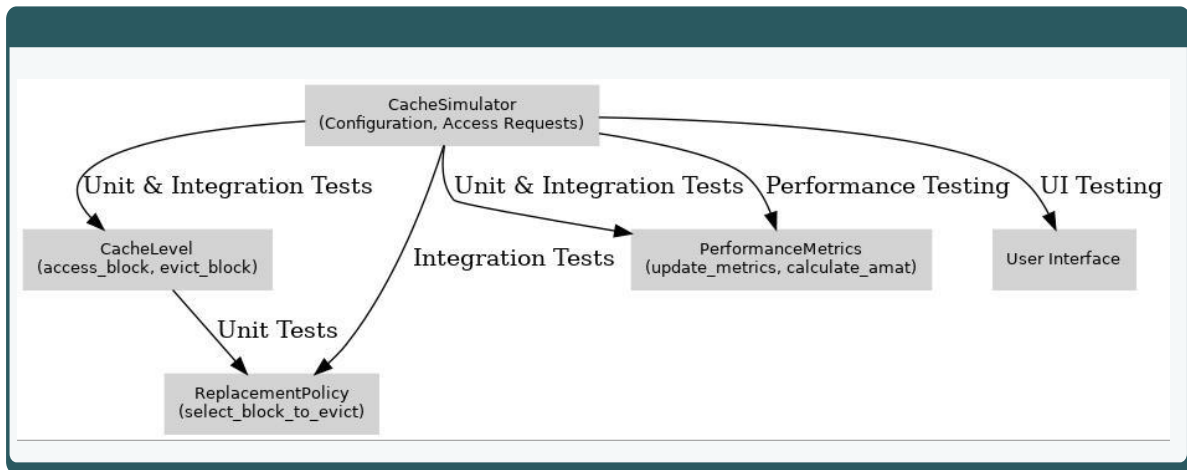


Figure: Testing Strategy

## 4.4. Planned Enhancements

With a solid foundation in place, future developments will extend the simulator's capabilities:

- **Multi-Level Cache Hierarchy Expansion:**
  - Addition of L3 as a fully associative cache, giving users a complete representation of real-world cache hierarchy.
  - Each level will be fully configurable to support different cache architectures and speeds.
- **Enhanced Visualization:**
  - Real-time graphs for hit/miss rate trends and AMAT, providing a visual representation of cache performance over time.
  - Detailed eviction information to show which blocks are removed and why, enhancing understanding of replacement policies.

- **Access Pattern Simulation:**
  - Built-in simulations for access patterns that emphasize spatial and temporal locality, enabling users to see how cache configurations adapt to these patterns.
- **Configurable Access Times:**
  - Setting distinct access times for each level will allow AMAT calculations to reflect realistic performance differences between L1, L2, and L3 caches.

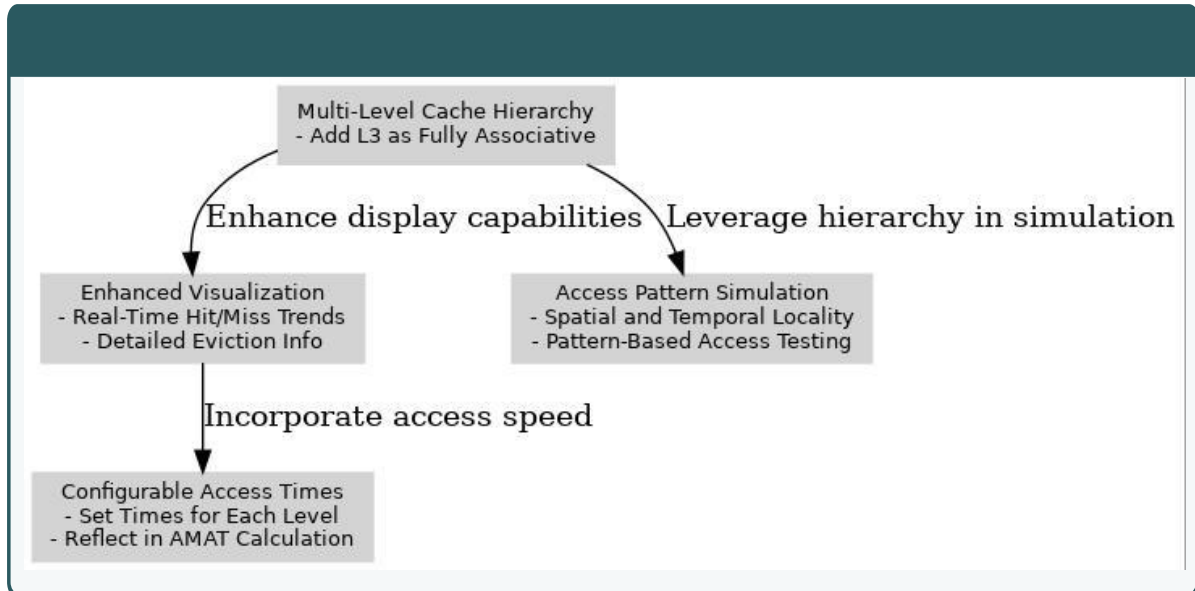


Figure: Planned Enhancements

## Implementation

### 5.1 Core Flask Application and API Routing: app.py

This file serves as the backend for the Cache Memory Simulator. It acts as a bridge between the frontend and the SetAssociativeCache class by exposing various endpoints to configure the cache, access memory, fetch statistics, and manage cache operations. The Flask framework is used to set up these endpoints.

#### File Description

The app.py file provides the following functionalities:

1. **System Configuration (/configure)**
  - Allows the user to configure the cache simulator with parameters such as cache size, block size, associativity, and replacement policy.
  - Validates input parameters to ensure they meet the simulator's requirements.
2. **Cache Access (/access)**
  - Supports both read and write operations to the cache.
  - Tracks cache hits and misses and updates the corresponding statistics.
3. **Batch Operations (/batch)**
  - Executes multiple read/write commands in a single batch.
  - Logs results for each operation (e.g., address, data, hit/miss) in the **History** section.
  - Efficiently visualizes the impact of bulk operations.

#### 4. Cache Operations

- Flush Cache: Clears all cached data and resets the state.
- Fetch Cache Statistics: Retrieves metrics like cache hits, misses, total accesses, miss rate, and AMAT.
- View Cache Contents: Provides a detailed view of the cache structure.
- Fetch Physical Memory: Displays the main memory content.

#### 5. Performance Metrics

- Tracks and computes **Average Memory Access Time (AMAT)**.
- Displays real-time performance metrics (Hits, Misses, AMAT) through a **graph** for better visualization.

#### 6. History Logging

- Logs all read, write, and batch operations, including details like operation type, address, data, and result (hit/miss).

## Key Functions of app.py

### 1. Configure the Cache System

```
@app.route('/configure', methods=['POST'])
def configure():
    global cache_system
    config = request.json

    # Validate configuration parameters
    address_width = config.get("address_width", 8)
    cache_size = config.get("cache_size")
    block_size = config.get("block_size")
    associativity = config.get("associativity", cache_size // block_size) # Fully
    associativity default
    replacement_policy = config.get("replacement_policy", "LRU")

    # Initialize the cache system
    try:
        cache_system = SetAssociativeCache(
            cache_size=cache_size,
            block_size=block_size,
            associativity=associativity,
            replacement_policy=replacement_policy,
            address_width=address_width,
        )
    except Exception as e:
        return jsonify({"error": str(e)}), 500
    return jsonify({"status": "Cache system configured successfully"})
```

*Code: Configure the Cache System*

- **Purpose:** Configures the cache simulator based on user input.

- **Key Features:**
  - Validates the input parameters.
  - Initializes a new instance of SetAssociativeCache.

## 2. Access the Cache

```
@app.route('/access', methods=['POST'])
def access():
    global cache_system
    if cache_system is None:
        return jsonify({"error": "Cache system not configured. Please configure the cache first."}), 400

    data = request.json
    address = int(data['address'])
    value = data.get('value') # Optional data for write operations

    # Access the cache and handle the result
    result = cache_system.access(address, data=value)

    return jsonify({
        "status": result["status"],
        "data": result["data"],
        "hits": cache_system.hits,
        "misses": cache_system.misses,
        "total_accesses": cache_system.total_accesses
    })
```

*Code: Access the Cache*

- **Purpose:** Handles read and write operations to the cache.
- **Key Features:**
  - Updates and returns cache statistics (hits, misses, total accesses).
  - Differentiates between read and write operations.



### 3. Fetch Cache Contents

```
@app.route('/cache_contents', methods=['GET'])
def cache_contents():
    global cache_system
    if cache_system is None:
        return jsonify({"error": "Cache system not configured. Please configure the system first."}), 400

    # Create a detailed view of cache contents
    cache_view = [
        [
            {
                "valid": block.valid,
                "dirty": block.dirty,
                "tag": block.tag,
                "data": block.data
            }
            for block in set_blocks
        ]
        for set_blocks in cache_system.cache
    ]

    return jsonify({
        "cache_contents": cache_view,
        "block_size": cache_system.block_size,
        "associativity": cache_system.associativity,
        "num_sets": cache_system.num_sets
    })
```

*Code: Fetch Cache Contents*

- **Purpose:** Provides a detailed view of the cache, including metadata like validity, tag, and data.
- **Key Features:**
  - Helps in debugging and understanding the current state of the cache.
  - Returns cache-level structural details such as block size, associativity, and number of sets.

This file provides the backbone for the Cache Memory Simulator by exposing essential endpoints. It manages the lifecycle of the cache system, facilitates memory access, and offers performance monitoring. The key snippets focus on configuration, cache access, and cache content retrieval, showcasing the most critical functionalities of the file.

## 5.2 Cache Logic and Core Functionality:

### cache\_simulator.py

This file contains the core implementation of the cache simulation, including the data structure for cache blocks and the SetAssociativeCache class. It handles cache operations like read/write access, block replacement, and tracking performance metrics, including Average Memory Access Time (AMAT).

## File Description

The `cache_simulator.py` file provides the following functionalities:

### 1. Cache Block Representation

- `CacheBlock` class models individual cache blocks, storing metadata such as validity, tag, and data.

### 2. Set-Associative Cache Implementation

- `SetAssociativeCache` class represents the cache structure, supporting configurable cache size, associativity, replacement policy, and block size.
- Implements cache operations including hit/miss handling, block replacement, and data storage.

### 3. Performance Metrics

- Tracks cache hits, misses, total accesses, and calculates Average Memory Access Time (AMAT).

## Key Components and Classes in `cache_simulator.py`

This file contains the core implementation of the cache simulation, including the data structure for cache blocks and the `SetAssociativeCache` class. It handles cache operations like read/write access, block replacement, and tracking performance metrics, including Average Memory Access Time (AMAT).

## File Description

The `cache_simulator.py` file provides the following functionalities:

### 1. Cache Block Representation

- `CacheBlock` class models individual cache blocks, storing metadata such as validity, tag, and data.

### 2. Set-Associative Cache Implementation

- `SetAssociativeCache` class represents the cache structure, supporting configurable cache size, associativity, replacement policy, and block size.
- Implements cache operations including hit/miss handling, block replacement, and data storage.

### 3. Performance Metrics

- Tracks cache hits, misses, total accesses, and calculates Average Memory Access Time (AMAT).

### 1. Cache Block Initialization

```
class CacheBlock:
    def __init__(self, block_size):
        self.valid = False
        self.tag = None
        self.data = [None] * block_size # Initialize as None to distinguish
from zero-initialized
        self.dirty = False
        self.last_access_time = 0 # For LRU replacement
        self.load_order = 0 # For FIFO replacement
```

*Code: Cache Block Initialization*

- **Purpose:** Represents individual cache blocks, storing metadata and data.
- **Key Features:**
  - **valid:** Indicates whether the block contains valid data.
  - **tag:** Stores the tag for identifying blocks.
  - **data:** Holds the actual data in the block.
  - **last\_access\_time** and **load\_order:** Used for replacement policies like LRU and FIFO.

## 2. Access Method

```
def access(self, address, data=None):
    """Access the cache with the given address and optionally store data."""
    self.total_requests += 1 # Increment total requests
    set_index = self.index(address)
    tag = self.tag(address)
    block_offset = address % self.block_size
    set_blocks = self.cache[set_index]

    # Check for cache hit
    for block in set_blocks:
        if block.valid and block.tag == tag:
            # Cache hit
            self.hits += 1
            self.total_access_time += self.access_time # Add access time for
hit
            if data is not None:
                block.data[block_offset] = data
                block.dirty = True
                block.last_access_time = self.load_counter
                self.load_counter += 1
                return {"status": "hit", "data": block.data}

    # Cache miss
    self.misses += 1
    self.total_access_time += self.access_time # Add access time for cache
access
    self.load_counter += 1

    # Handle cache miss - find a block to replace
    block_to_replace = self.find_replacement_block(set_blocks)

    # Load data from physical memory into the block
    base_address = address - block_offset # Align to block start
    block_to_replace.data = self.memory[base_address:base_address +
self.block_size]
    block_to_replace.valid = True
    block_to_replace.tag = tag
    block_to_replace.last_access_time = self.load_counter
    block_to_replace.dirty = False

    # Write data if this is a write operation
    if data is not None:
        block_to_replace.data[block_offset] = data
        block_to_replace.dirty = True

    self.total_access_time += self.memory_access_time # Add memory access time
for miss
    return {"status": "miss", "data": block_to_replace.data}
```

*Code: Access Method*

- **Purpose:** Handles cache access for read/write operations.
- **Key Features:**
  - Tracks hits and misses.
  - Manages data loading from physical memory on cache misses.
  - Updates AMAT-related metrics like `total_access_time` and `total_requests`.

This file forms the backbone of the cache simulation by defining the data structures and operations necessary for the simulator. The chosen snippets highlight the core functionality of cache block representation, cache access, and AMAT computation.

## 5.3 Frontend Logic and API Interaction: `script.js`

This file contains the JavaScript code that powers the dynamic and interactive features of the cache simulation web application. It handles user interactions, communicates with the backend API, and updates the UI with the results of cache operations and performance metrics. The `script.js` file provides the following functionalities:

- **Dynamic UI Updates**
  - Renders cache contents, physical memory, and performance metrics.
  - Updates performance graphs and displays real-time metrics like cache hits, misses, and AMAT.
- **API Communication**
  - Initializes the cache system via API calls for configuration, memory access, batch operations, and flushing.
  - Retrieves and updates cache data, physical memory, and statistics.
- **Performance Metrics**
  - Displays hits, misses, and AMAT in real-time and visualizes trends with an interactive graph.
- **User Interactions**
  - Enables manual and batch cache operations with results logged in the History section.
  - Supports cache flushing to reset the system and clear metrics, tables, and graphs.

### Key Functions and Structure in `script.js`

- **Purpose:** Dynamically updates the cache size dropdown based on the selected cache level.
- **Key Features:**
  - Ensures the UI reflects valid cache size options for L1, L2, and L3 levels.

- Simplifies user input and prevents invalid configurations.

## 2. Cache Access Handling

```
async function accessMemory(mode) {
  const address = parseInt(document.getElementById("address").value);
  const value = mode === 'write' ?
    parseInt(document.getElementById("data").value) : null;

  const response = await fetch('/access', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ address, value })
  });

  const result = await response.json();
  const resultElement = document.getElementById("result");

  if (result.status === "hit") {
    resultElement.innerText = `Cache Hit! Data: ${result.data}`;
  } else if (result.status === "miss") {
    resultElement.innerText = `Cache Miss! Loaded Data: ${result.data}`;
  } else if (result.error) {
    alert(result.error);
  }
}
```

*Code: Cache Access Handling*

- **Purpose:** Handles cache read/write operations based on user input.
- **Key Features:**
  - Communicates with the backend to perform cache operations.
  - Updates the UI with the result of each operation (hit/miss).

This file bridges the gap between the backend and the frontend, ensuring that the UI remains responsive and dynamic. The chosen snippets illustrate the file's core functionalities: configuring cache levels, handling access operations, and displaying performance metrics.

## 5.4 User Interface Structure: index.html

This file is the main user interface of the Cache Simulator application. It provides a structured layout for configuring the cache, accessing memory, viewing cache/memory contents, and monitoring performance metrics like hits, misses, and AMAT.

### File Description

The index.html file accomplishes the following:

#### 1. Configuration Section

- Allows users to set the address width, cache level, cache size, block size, associativity, and replacement policy.
- Provides a button to generate the cache system based on these parameters.
- 2. **Manual Access**
  - Includes fields for manual read/write operations to a specific memory address.
  - Displays the result of cache accesses (hit or miss) and highlights affected cells in the cache or memory.
- 3. **Cache and Memory Visualization**
  - Dynamically populates tables showing the cache contents and physical memory after operations.
  - Highlights cells affected by read/write operations for better user understanding.
- 4. **Performance Metrics**
  - Displays cache statistics such as hits, misses, and AMAT.
  - Provides per-level cache statistics (for multi-level caches).
- 5. **History Section**
  - Logs all user interactions (e.g., reads, writes) for reference.

## Key Components in index.html

### 1. Configuration Section

```
<label for="cacheSize">Cache Size:</label>
<select id="cacheSize">
  <!-- Dynamically populate based on cache level -->
</select> <button onclick="configureCache()">Generate System</button>
```

*Code: Configuration Section*

- **Purpose:** Allows users to configure cache parameters such as level and size, and generate the cache system.
- **Key Feature:**
  - Dropdowns for dynamic configuration.

## 2. Performance Metrics Section

```
<div id="performanceMetrics" class="cache-stats">
  <h2>Performance Metrics</h2>
  <div id="amat">Average Memory Access Time (AMAT): <span
id="amatValue">N/A</span></div>
  <div id="perLevelStats">
    <h3>Per-Level Cache Statistics</h3>
    <ul id="levelStats">
      <!-- Per-level stats will be dynamically populated -->
    </ul>
  </div>
</div>
```

*Code: Performance Metrics Section*

- **Purpose:** Displays real-time performance metrics, including AMAT and per-level statistics.
- **Key Feature:**
  - Dynamically updated using JavaScript to reflect the latest performance data.

## 3. Cache Contents Visualization

```
<div class="cache-grid">
  <h2>Cache Contents (VDT Cache Data)</h2>
  <div class="memory-scroll-box" style="height: 200px; overflow-y: auto;
border: 1px solid #ccc;">
    <table id="cacheTable" class="memory-table">
      <!-- Headers will be generated dynamically -->
    </table>
  </div>
</div>
```

*Code: Cache Contents Visualization*

- **Purpose:** Provides a visual representation of cache contents, including valid bits, tags, and data.
- **Key Feature:**
  - Supports dynamic updates to reflect the current state of the cache after each operation.

The `index.html` file serves as the backbone of the Cache Simulator's user interface. It enables users to configure, interact with, and analyze the cache system in an intuitive and visually appealing way. The chosen snippets demonstrate the file's core functionalities, focusing on configuration, performance metrics, and visualization.

## 5.5 User Interface Styling: `style.css`

The stylesheet defines the visual layout and design of the Cache Simulator web application. It ensures that the user interface (UI) is visually appealing, functional, and responsive.

## File Description

The CSS file customizes the UI elements for the following components:

1. **General Layout**
  - Applies consistent typography, spacing, and padding across the application.
  - Defines global styles for headings, buttons, inputs, and containers.
2. **Cache and Memory Visualization**
  - Adds styles for the cache and memory tables, including cell highlighting for hits and misses.
  - Provides a scrollable layout for large data tables (e.g., memory contents).
3. **Performance Metrics**
  - Styles the Average Memory Access Time (AMAT) and per-level cache statistics sections.
  - Highlights important metrics to improve user readability.
4. **Interactive Elements**
  - Adds hover effects for buttons and table rows.
  - Includes custom highlighting for selected rows and cells.

## Key Styling Elements

### 1. Highlighting Hits and Misses

```
.result.hit {  
  background-color: #d4edda;  
  color: #155724;  
}  
  
.result.miss {  
  background-color: #f8d7da;  
  color: #721c24;  
}
```

*Code: Highlighting Hits and Misses*

- **Purpose:** Visually distinguishes between cache hits and misses in the results section.
- **Key Feature:**
  - Green background for hits.
  - Red background for misses.

### 2. Scrollable Memory and Cache Tables



```

.memory-scroll-box {
  height: 200px;
  max-height: 200px;
  overflow-y: auto;
  overflow-x: hidden;
  background-color: #f9f9f9;
  border: 1px solid #ccc;
  margin-top: 10px;
}

.memory-table th,
.memory-table td {
  border: 1px solid #ccc;
  padding: 5px;
  text-align: center;
  font-size: 14px;
}

```

*Code: Scrollable Memory and Cache Tables*

- **Purpose:** Ensures memory and cache tables remain readable and fit within the screen without clutter.
- **Key Feature:**
  - Adds vertical scrolling for large datasets.
  - Maintains uniform cell sizes and font consistency.

### 3. Performance Metrics

```

#amat {
  text-align: center;
  padding: 10px;
  background-color: #ffffbf0;
  border-radius: 4px;
  border: 1px solid #f0e0c0;
}

#per-level-stats ul li {
  padding: 5px 0;
  border-bottom: 1px solid #eee;
}

```

*Code: Performance Metrics*

- **Purpose:** Visually emphasizes the AMAT and per-level stats to improve clarity.
- **Key Feature:**
  - Distinct yellow background for AMAT to make it stand out.
  - Cleanly formatted list for per-level statistics.

This stylesheet ensures that the Cache Simulator is user-friendly and visually structured. It focuses on readability and interactivity while enhancing the overall aesthetic of the web application. The

chosen snippets highlight key elements such as hits/misses, scrollable tables, and performance metrics.

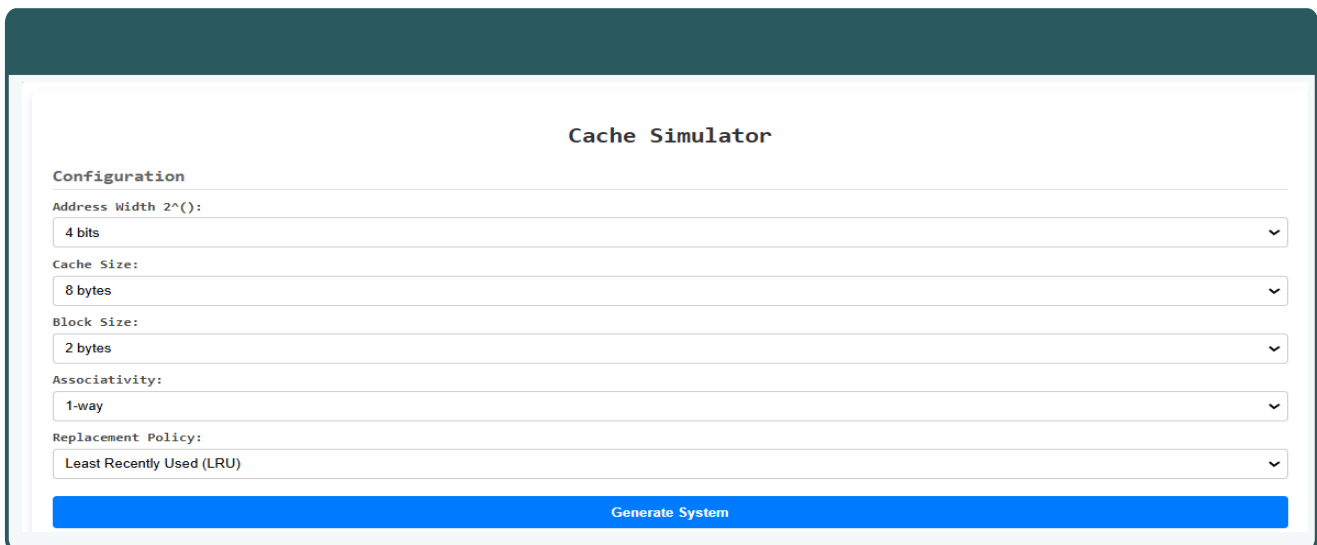
## 5.6 User Manual

The Cache Memory Simulator interface is divided into several sections, making it straightforward for users to configure, interact with, and observe cache operations. Below is a step-by-step guide to using each section of the interface.

### 5.6.1 Configuration Section

1. **Address Width:** Select the number of bits for the address. This parameter determines the addressable memory space.
2. **Cache Size:** Choose the total size of the cache in bytes. This size determines the total memory that can be stored in the cache.
3. **Block Size:** Select the size of each cache block. The block size impacts how many data bytes are fetched together for each memory request.
4. **Associativity:** Set the associativity of the cache:
  - 1-way represents a **direct-mapped** cache.
  - Higher levels (e.g., 2-ways, 4-ways) represent **set-associative** caches.
5. **Replacement Policy:** Choose the policy for evicting blocks from the cache. Options include:
  - **LRU** (Least Recently Used): Evicts the block that has not been accessed for the longest time.
  - **FIFO** (First-In-First-Out): Evicts blocks in the order they were loaded.
  - **Random:** Evicts a random block when the cache is full.

After configuring these parameters, click the **Generate System** button to initialize the cache with settings. The system will now be ready for interaction.



The screenshot shows the 'Cache Simulator' configuration interface. It features a title bar 'Cache Simulator' and a 'Configuration' section. The configuration section contains six dropdown menus: 'Address Width 2^():' set to '4 bits', 'Cache Size:' set to '8 bytes', 'Block Size:' set to '2 bytes', 'Associativity:' set to '1-way', 'Replacement Policy:' set to 'Least Recently Used (LRU)', and a blue 'Generate System' button at the bottom.

*Interface: Part 1*

## 5.6.2 Manual Access Section

- **Address Field:** Enter a memory address that you want to read from or write to in the cache.
- **Data Field:** Enter data to write into the cache at the specified address (for write operations only).

### Buttons:

- **Read:** Simulates a read operation on the cache. If the address is already in the cache, it will result in a **cache hit**; otherwise, it will be a **miss**.
- **Write:** Writes the specified data to the address in the cache. If the address is not in the cache, it will be fetched, possibly evicting another block if necessary.
- **Flush:** Clears all data from the cache, resetting it to an empty state.



The screenshot shows a web interface titled "Manual Access". It contains two input fields: "Address:" with a placeholder "Enter address" and "Data:" with a placeholder "Enter data (for writes)". Below these fields are three blue buttons labeled "Read", "Write", and "Flush".

*Interface: Part 2*

## 5.6.3 Batch Operations Section

The Batch Operations feature allows users to perform multiple cache operations simultaneously. Users can enter a list of commands, each on a new line, specifying the type of operation (READ or WRITE), the address, and optional data for WRITE operations.

### How to Use

1. **Input Commands:**
  - Enter each command in the provided text area, following the format:
    - READ <address> (e.g., READ 0x0)
    - WRITE <address> <data> (e.g., WRITE 0x3 0x4)
  - Use hexadecimal format for addresses and data values.
2. **Run Batch:**
  - Click the **Run Batch** button to execute all entered commands sequentially.
3. **View Results:**
  - Results for each operation appear below the button, detailing:
    - Operation type (READ/WRITE).
    - Address accessed.
    - Data written (for WRITE operations).
    - Result (HIT or MISS).

### Key Benefits

- Simplifies the process of testing multiple cache scenarios.

- Allows users to analyze the cache's behavior under varied access patterns efficiently.
- Results are displayed in real-time, aiding quick debugging and learning.

Batch Operations

Read 0  
Write 3 4

Run Batch

Operation: READ Address: 0x0 Result: miss

Operation: WRITE Address: 0x3 Data: 0x4 Result: miss

Interface: Part 3

## 5.6.4 Cache Statistics Section

This section provides real-time updates on:

- **Cache Hits:** The number of times data was found in the cache.
- **Cache Misses:** The number of times data was not in the cache and had to be fetched from memory.

Cache Statistics

Cache Hits: 0

Cache Misses: 0

Interface: Part 4

## 5.6.5 Cache Contents (VDT Cache Data)

Displays the current contents of the cache:

- **Set** and **Block:** Indicate the specific set and block within the cache.
- **V** (Valid), **D** (Dirty), and **T** (Tag) columns provide metadata about each cache block.
- **Bytes** columns show the data stored in each block.

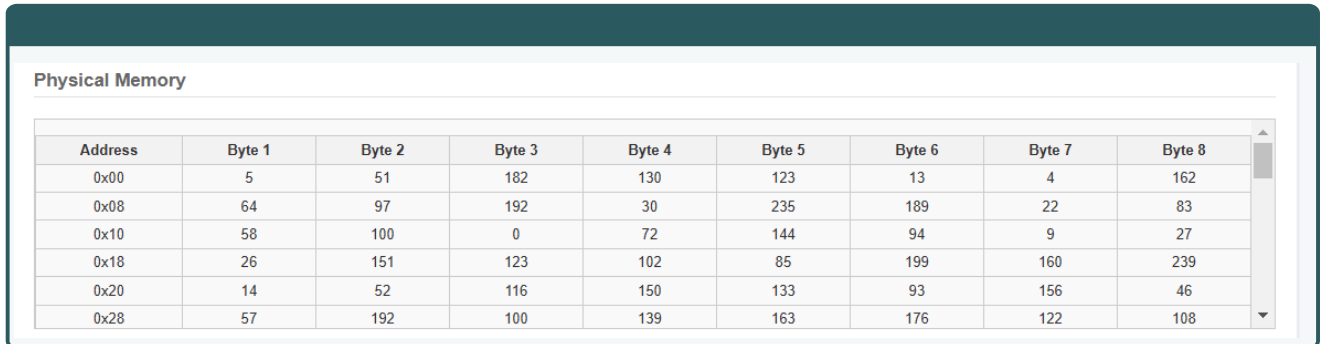
Cache Contents (VDT Cache Data)												
Set	Block	V	D	T	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
0	0	false	false	null	null	null	null	null	null	null	null	null
1	0	false	false	null	null	null	null	null	null	null	null	null
2	0	false	false	null	null	null	null	null	null	null	null	null
3	0	false	false	null	null	null	null	null	null	null	null	null

Interface: Part 5

## 5.6.6 Physical Memory

This section shows a snapshot of the main memory, allowing users to compare cache contents with actual memory data. It includes:

- **Address:** The main memory address.
- **Byte Columns:** Data stored at each byte of memory.



The screenshot shows a web interface titled "Physical Memory". It contains a table with 9 columns: "Address", "Byte 1", "Byte 2", "Byte 3", "Byte 4", "Byte 5", "Byte 6", "Byte 7", and "Byte 8". The table has 7 rows of data. A vertical scrollbar is visible on the right side of the table.

Address	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
0x00	5	51	182	130	123	13	4	162
0x08	64	97	192	30	235	189	22	83
0x10	58	100	0	72	144	94	9	27
0x18	26	151	123	102	85	199	160	239
0x20	14	52	116	150	133	93	156	46
0x28	57	192	100	139	163	176	122	108

*Interface: Part 6*

## 5.6.7 Performance Metrics and Graphs

This section provides real-time visual feedback on the cache's performance through both numerical data and an interactive graph.

### Average Memory Access Time (AMAT)

- Displays the calculated **Average Memory Access Time (AMAT)**, which represents the average time required to retrieve data from memory.
- AMAT dynamically updates based on cache configuration, operations, and results (hits and misses).

### Performance Metrics Graph

- The graph dynamically visualizes the following metrics over a series of operations:
  - **Cache Hits:** Number of successful data accesses from the cache.
  - **Cache Misses:** Number of unsuccessful data accesses requiring retrieval from main memory.
  - **AMAT:** Shows how the Average Memory Access Time evolves with each operation.
- **Graph Features:**
  - Real-time updates after every cache operation.
  - Smooth curves for clear interpretation of trends.
  - Individual data points for each operation (e.g., Op1, Op2) for detailed analysis.

### How to Use

1. Perform cache operations using manual inputs or batch processing.
2. Observe how hits, misses, and AMAT are plotted on the graph in real time.
3. Analyze how the cache's efficiency varies with different configurations and workloads.

## Key Benefits

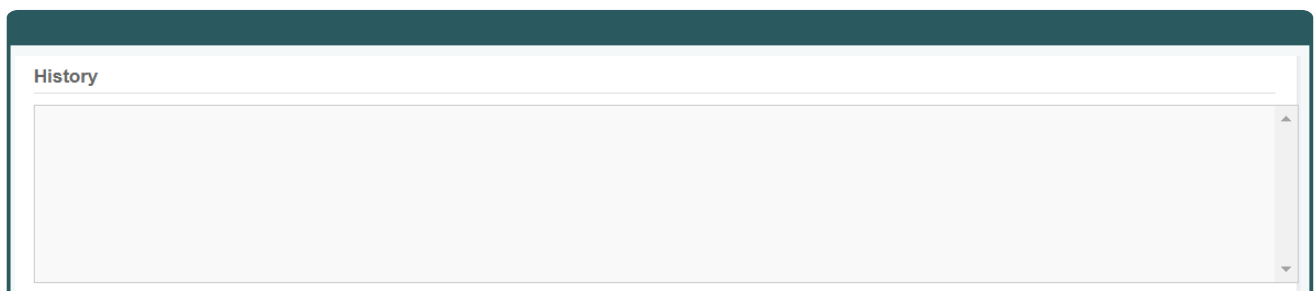
- Provides immediate insights into the impact of user actions on cache performance.
- Encourages experimentation by showing clear visual trends, aiding learning and debugging.



Interface: Part 7

## 5.6.8 History

The History section logs all operations (reads and writes), showing the sequence of actions performed by the user and whether each action resulted in a cache hit or miss. This feature allows users to analyze cache behavior over time.



Interface 5.6.6: Part 8

# Summary

The Cache Memory Simulator project offers a comprehensive platform to explore and analyze the intricacies of cache memory systems. It supports direct-mapped, set-associative, and fully associative caches, providing users with the flexibility to experiment with a wide range of cache configurations. Key features include the ability to select multiple replacement policies (e.g., LRU, FIFO, Random), real-time visualization of cache behavior, batch processing of operations, and configurable multi-level caching to simulate real-world systems. The simulator also tracks detailed performance metrics such as cache hit/miss rates and Average Memory Access Time (AMAT), providing valuable insights into the practical impact of various cache design choices.

# Conclusion

The Cache Memory Simulator successfully achieves its goal of bridging theoretical concepts and practical applications in computer architecture. By implementing direct-mapped, set-associative, and fully associative caches, along with a range of replacement policies, the project provides users with a versatile tool to understand and optimize cache behavior. Features such as real-time visualization, performance metrics tracking, and multi-level caching emulate real-world systems, making the simulator an invaluable resource for education and research. The simulator meets all objectives and offers a strong foundation for further exploration and enhancements in cache memory design and analysis.

# Bibliography

1. Patterson, David A., and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 5th ed., Morgan Kaufmann, 2014.
2. Stallings, William. *Computer Organization and Architecture: Designing for Performance*. 9th ed., Pearson, 2013.
3. Smith, Alan Jay. "Cache Memories." *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, 1982, pp. 473-530.
4. Hill, Mark D., and Norman P. Jouppi. "A Unified View of Cache Performance." *Proceedings of the 18th Annual International Symposium on Computer Architecture*, IEEE, 1991, pp. 161-170.
5. Hennessy, John L., and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th ed., Elsevier, 2017.
6. <https://witscad.com/course/computer-architecture/chapter/cache-memory#3Cs-of-misses>
7. [https://www.researchgate.net/figure/A-classical-three-level-cache-hierarchy\\_fig1\\_362707415](https://www.researchgate.net/figure/A-classical-three-level-cache-hierarchy_fig1_362707415)
8. <https://dev.to/satrobot/cache-replacement-algorithms-how-to-efficiently-manage-the-cache-storage-2ne1>