

**Cache Simulator with Replacement Policies**

***Group Members***

Ethem Kesim 210302205

Emre Çubuk 220302393

Ozan Umut Güney 200302109

# ***Contents***

[***Contents*** 2](#_Toc199766462)

[***1. Introduction*** 2](#_Toc199766463)

[***1.1 Project Objective*** 2](#_Toc199766464)

[**1.2 Overview** 3](#_Toc199766465)

[***2. Background*** 3](#_Toc199766466)

[***2.1 Cache Memory Fundamentals*** 3](#_Toc199766467)

[***2.2 Cache Organization*** 3](#_Toc199766468)

[***2.3 Replacement Policies*** 3](#_Toc199766469)

[***2.4 Cache Performance Metrics*** 4](#_Toc199766470)

[***3. Project Design and Architecture*** 4](#_Toc199766471)

[***3.1 Overview*** 4](#_Toc199766472)

[***3.2 Components*** 4](#_Toc199766473)

[***3.2.1Cache Simulator Core (cache\_simulator.py)*** 4](#_Toc199766474)

[***3.2.2 Trace Generator (generate\_trace.py)*** 4](#_Toc199766475)

[***3.2.3 Analysis and Visualization (main.ipynb)*** 4](#_Toc199766476)

[***3.3 Design Decisions*** 5](#_Toc199766477)

[***3.4 Data Flow*** 5](#_Toc199766478)

[***4. Implementation and Code Review*** 5](#_Toc199766479)

[***4.1 generate\_trace.py*** 5](#_Toc199766480)

[***4.2 cache\_simulator.py*** 7](#_Toc199766481)

[***4.3 main.ipynb*** 11](#_Toc199766482)

[***5.*** ***Experimental Results*** 14](#_Toc199766483)

[***5.1 Setup Summary*** 14](#_Toc199766484)

[***5.2 Results Overview*** 15](#_Toc199766485)

[***5.3 Example Findings*** 15](#_Toc199766486)

[***5.4 Cold vs. Conflict Misses*** 15](#_Toc199766487)

[***6.*** ***Challenges Faced*** 16](#_Toc199766488)

[***7. Conclusion*** 17](#_Toc199766489)

# ***1. Introduction***

## ***1.1 Project Objective***

The main objective of this project is to develop a cache memory simulator that models different cache configurations and replacement policies. By simulating various scenarios, the project aims to analyze the impact of cache size, block size, associativity, and replacement policies on cache performance. This allows us to understand how these factors influence hit and miss rates in a cache system.

## **1.2 Overview**

Cache memory plays a crucial role in modern computer architecture by reducing the average time to access data from the main memory. Different cache designs and replacement strategies affect the efficiency of caching mechanisms. This project implements a simulator that generates memory access traces and evaluates cache performance under different configurations such as LRU, FIFO, and Random replacement policies.

The simulation results are presented through detailed logs, statistical summaries, and graphical visualizations to provide clear insights into cache behavior and performance trade-offs.

# ***2. Background***

## ***2.1 Cache Memory Fundamentals***

Cache memory is a small, fast memory located close to the CPU that stores copies of frequently accessed data from the main memory. Its purpose is to reduce the latency of data access and improve overall system performance. When the CPU requests data, the cache is checked first. If the data is found (a *cache hit*), it is delivered quickly. If not (a *cache miss*), the data must be fetched from slower main memory, causing delays.

## ***2.2 Cache Organization***

Caches are organized into blocks (or lines), which are the smallest units of data transfer between the cache and main memory. These blocks are grouped into sets, depending on the associativity of the cache:

* **Direct-mapped cache**: Each memory block maps to exactly one cache line.
* **Set-associative cache**: Each memory block can map to any cache line within a set.
* **Fully associative cache**: A memory block can be placed anywhere in the cache.

The associativity affects the flexibility and complexity of cache replacement.

## ***2.3 Replacement Policies***

When the cache is full and a new block needs to be loaded, the cache must decide which existing block to evict. Common replacement policies include:

* **LRU (Least Recently Used)**: Evicts the block that has not been used for the longest time.
* **FIFO (First In, First Out)**: Evicts the oldest block in the cache.
* **Random**: Evicts a randomly selected block.

Each policy has trade-offs in terms of implementation complexity and performance.

## ***2.4 Cache Performance Metrics***

Cache performance is typically evaluated by measuring:

* **Hit Rate**: The proportion of memory accesses found in the cache.
* **Miss Rate**: The proportion of memory accesses not found in the cache.
* **Miss Types**:
  + *Cold Misses*: Occur when a block is accessed for the first time.
  + *Conflict Misses*: Occur when multiple blocks compete for the same cache set.

Understanding these metrics helps in optimizing cache design and application performance.

# ***3. Project Design and Architecture***

## ***3.1 Overview***

The cache simulator project is designed to analyze and evaluate the performance of various cache configurations and replacement policies. The architecture is modular to separate concerns and allow easy modification and extension.

## ***3.2 Components***

### ***3.2.1Cache Simulator Core (cache\_simulator.py)***

* + Implements the core logic of the cache simulation.
  + Configurable parameters include cache size, block size, associativity, and replacement policy.
  + Tracks cache hits, misses, and detailed miss types (cold misses, conflict misses).
  + Uses data structures like deque for efficient cache set management.

### ***3.2.2 Trace Generator (generate\_trace.py)***

* + Responsible for generating memory access traces that simulate realistic access patterns with temporal and spatial locality.
  + Adjustable parameters allow customization of trace length, base address, and locality window.
  + Ensures that each simulation run can have varying trace data to analyze cache behavior under different scenarios.

### ***3.2.3 Analysis and Visualization (main.ipynb)***

* + Orchestrates multiple simulation runs over different cache configurations.
  + Collects results such as hit and miss ratios.
  + Outputs results to CSV files for record-keeping.
  + Generates comparative bar charts to visualize the impact of different cache parameters and policies on performance.

## ***3.3 Design Decisions***

* **Modularity:** Separating simulation logic, trace generation, and analysis allows independent development and testing of each component.
* **Configurability:** All major cache parameters are easily adjustable, facilitating experimentation with different cache architectures.
* **Performance Tracking:** Beyond simple hit/miss statistics, the simulator categorizes misses into cold and conflict misses to provide deeper insight into cache behavior.
* **Reusability:** The simulator is built to be reused for various trace files and configurations without code changes, promoting flexibility.

## ***3.4 Data Flow***

1. The **Trace Generator** produces a memory access trace file.
2. The **Cache Simulator** loads the trace and simulates cache behavior according to the given configuration.
3. Simulation results are recorded, including detailed statistics.
4. The **Analysis Notebook** runs multiple simulations, aggregates results, saves them, and visualizes performance metrics.

# ***4. Implementation and Code Review***

In this section, we will review the implementation of the cache simulator by examining the core Python modules used. Each component is explained with its purpose, structure, and how it contributes to the overall simulation.

## ***4.1 generate\_trace.py***

metin, yazı tipi, ekran görüntüsü, beyaz içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

**random:** Provides functions for generating random numbers, essential for creating varying memory access patterns.

**time:** Used here to seed the random number generator with the current system time, ensuring different outputs on each execution.



This function generates a trace file of virtual memory addresses.

Parameters:

* **filename:** The name of the output trace file.
* **length:** Number of memory access entries.
* **base\_addr:** Starting base address for memory simulation.
* **locality\_window:** Defines the range for spatial locality.



* Seeds the random number generator with the current system time.
* Ensures that each time the script is run, a different trace file is generated, enhancing the variability and realism of the simulation.

metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

* Generates each memory access entry:
  + 50% chance to pick an address close to the **base\_addr** (simulating spatial locality).
  + 50% chance to pick a more distant address (introducing randomness).
* This mix mimics how programs often access memory: mostly local addresses with occasional jumps.

metin, yazı tipi, çizgi, beyaz içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

* Writes the generated memory addresses to the specified trace file in hexadecimal format, line by line.
* This file is later used as input by the cache simulation system.

metin, yazı tipi, çizgi, ekran görüntüsü içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

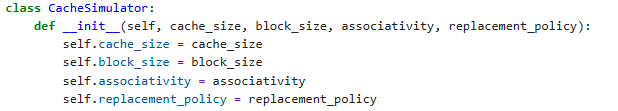
* Automatically calls the function when the script runs.
* Generates the example\_trace.txt file with 1000 memory access entries.

## ***4.2 cache\_simulator.py***

metin, yazı tipi, ekran görüntüsü, beyaz içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

* **random:** Used for the **Random replacement policy**.
* **deque:** A double-ended queue from the collections module, efficiently supports adding/removing elements for LRU and FIFO policies.
* **os:** Ensures log files can be written by creating necessary directories if they don't exist.



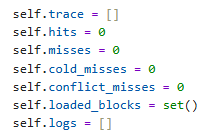
This method initializes a new cache simulator instance.

**Parameters**:

* **cache\_size:** Total size of the cache (in blocks).
* **block\_size:** Size of each block.
* **associativity:** Number of blocks per set (e.g., 1 for direct-mapped, N for N-way set associative).
* **replacement\_policy:** One of "LRU", "FIFO", or "Random".



* **num\_sets** is calculated based on associativity.
* **self.cache** is a list of sets (each as a deque), representing the entire cache.



These counters track performance metrics.

* **Cold Misses**: First-time accesses (compulsory).
* **Conflict Misses**: Repeated accesses that map to the same set but exceed associativity.
* **loaded\_blocks**: Helps distinguish cold vs. conflict misses.

metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

* Loads memory addresses (hex format) from a trace file into the simulator.
* Converts hex addresses into integers.



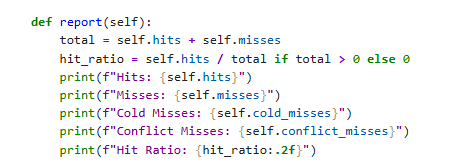
* Determines which cache **set** an address maps to.
* Important for implementing **set-associative** behavior.



This method processes each memory access from the trace and simulates the cache behavior step by step.

**Key logic:**

* Converts an address into a **block number** and **set index**.
* Checks if the block is in the corresponding cache set:
  + If present → **Cache Hit** (update hits, log result).
    - For LRU, move block to the end (most recently used).
  + If absent → **Cache Miss**:
    - Increment misses.
    - If block never loaded before → **Cold Miss**.
    - Otherwise → **Conflict Miss**.
    - If set is full:
      * Remove block based on the chosen replacement policy:
        + FIFO → Remove oldest block.
        + Random → Remove any block.
        + LRU → Remove least recently used.
    - Add new block to the set.



* Prints summary statistics:
  + Total hits and misses.
  + Breakdown of cold vs. conflict misses.
  + Hit ratio.

This is useful for debugging and analyzing cache performance manually.

metin, yazı tipi, ekran görüntüsü, çizgi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

* Saves simulation logs to a specified file.
* Automatically creates the directory if it does not exist using os.makedirs.

## ***4.3 main.ipynb***



* **CacheSimulator**: The core simulation engine imported from an external Python module.
* **matplotlib.pyplot**: Used for drawing performance bar charts of hit/miss ratios.

metin, yazı tipi, ekran görüntüsü içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

* This list contains **tuples** representing various cache setups to test:
  + **cache\_size:** Number of blocks in the cache (e.g., 16, 32, 64).
  + **block\_size:** Set to 1 in all tests for simplicity.
  + **associativity:** Number of blocks per set (1, 2, or 4).
  + **policy:** Replacement policy (LRU, FIFO, or Random).

Total Configurations Tested: 21 (3 associativities × 3 policies × 3 sizes)

metin, yazı tipi, beyaz, çizgi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

* This block initializes a CSV file to store each configuration’s **hit and miss ratios** for external analysis or comparison.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

* **For each configuration**, a new CacheSimulator instance is created and run using the same trace file (example\_trace.txt).
* If the trace file is missing, it skips that test and prints an error message.



* **hit\_ratio** and **miss\_ratio** are computed from the simulation results.

metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

* **label:** This line creates a unique string identifier for each configuration in the format

{cache\_size}-blk{block\_size}-assoc{associativity}-{policy}

For example:  
16-blk1-assoc2-LRU

* **results.append(...)**  
  This appends a **tuple** to the results list containing:
* the configuration label,
* the hit ratio,
* the miss ratio.
* **out\_csv.write(...)**
* Each result is also **written to the CSV file** so that it can be reviewed or processed later (e.g., in Excel).
* **sim.save\_logs(...)**
* This stores detailed per-access logs into a .log file inside the logs/ directory, named after the configuration label

**metin, yazı tipi, ekran görüntüsü, beyaz içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.**

* This block **extracts the stored data** (label, hit\_ratio, miss\_ratio) into separate lists:
  + **labels:** For X-axis tick labels.
  + **hit\_ratios:** Bar heights (green).
  + **miss\_ratios:** Bar heights (red), stacked on top of hit bars.

metin, yazı tipi, ekran görüntüsü, çizgi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

* **plt.subplots(...)** creates a large figure for better readability.
* **Green bars** represent the **hit ratios**.
* **Red bars** are **stacked on top** of the green bars (using bottom=hit\_ratios) to show **total access**.

metin, yazı tipi, ekran görüntüsü, çizgi içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

* Y-axis shows the **ratio scale** (from 0 to 1).
* X-axis displays **each configuration** label.
* **rotation=90** tilts the labels vertically for better spacing.
* **tight\_layout()** avoids label overlap.
* **plt.show()** finally renders the chart.

## ***Experimental Results***

The experimental results were obtained by simulating a variety of cache configurations using the custom cache simulator. Each configuration varied in terms of cache size, block size, associativity, and replacement policy (LRU, FIFO, Random). A synthetic trace file was used as input, and performance was evaluated primarily using the **cache hit ratio** and **miss ratio**.

### ***5.1 Setup Summary***

* **Trace File**: Randomly generated memory access pattern (example\_trace.txt)
* **Configurations Tested**:
* Cache Sizes: 16, 32, 64 blocks
* Block Size: 1
* Associativity: 1, 2, 4
* Replacement Policies: LRU, FIFO, Random
* **Metrics Recorded**:
* Total cache hits
* Total cache misses
* Cold misses and conflict misses
* Hit ratio and miss ratio
* Logs saved for each configuration

### ***5.2 Results Overview***

The results are saved in the results.csv file and visualized as a bar chart. The green bars indicate the hit ratio, while red bars stacked on top indicate the corresponding miss ratio. Each bar group corresponds to a unique cache configuration.

The bar chart demonstrates clear trends:

* Increasing the **cache size** generally improves the hit ratio, as there is more space to store blocks.
* Increasing **associativity** helps reduce **conflict misses**, especially with smaller caches.
* **LRU** typically performs slightly better than FIFO or Random, particularly at higher associativity levels.

### ***5.3 Example Findings***

* 32-blk1-assoc2-LRU showed a significantly higher hit ratio than 16-blk1-assoc1-Random.
* 64-blk1-assoc1-LRU resulted in some of the highest hit ratios, confirming that both larger size and intelligent replacement policies are beneficial.
* FIFO and Random policies were less effective in configurations with higher conflict rates.

### ***5.4 Cold vs. Conflict Misses***

From the simulation logs and internal counters:

* **Cold Misses** occurred only the first time a block was loaded.
* **Conflict Misses** occurred when a block was evicted prematurely due to mapping conflicts in low-associativity caches.
* As associativity increased, the number of **conflict misses decreased**, validating the effectiveness of set-associative caches.

metin, menü, sayı, numara, ekran görüntüsü içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

metin, menü, ekran görüntüsü, sayı, numara içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

## ***Challenges Faced***

**Trace File Randomization**

A major challenge was ensuring that the trace file produced diverse and dynamic memory access patterns on each execution. To achieve this, randomness was incorporated using Python’s random module seeded with the current system time. This approach helped simulate both temporal and spatial locality in access patterns. However, ensuring that the variation still yielded meaningful comparisons across cache configurations required balancing between randomness and consistency.

**Code Integration and Output Management**

Another challenge was integrating the Python simulation logic into a clean and modular Jupyter Notebook workflow. Importing external Python modules (like cache\_simulator.py) into the notebook, maintaining correct file paths, and handling output directories (e.g., logs and CSVs) without causing file I/O errors required careful coordination. Additionally, improving the readability of performance results and visual outputs demanded custom formatting, structured CSV exports, and clean visualization design.

## ***7. Conclusion***

This project aimed to design and implement a configurable cache simulator to explore the impact of various architectural parameters — such as cache size, block size, associativity, and replacement policies — on overall cache performance.

Through the generation of synthetic memory access traces and the simulation of multiple cache configurations, we were able to:

* Quantify the effects of cold and conflict misses,
* Compare different replacement policies (LRU, FIFO, Random),
* Visualize hit/miss ratios across setups,
* And understand the importance of associativity in reducing cache conflicts.

The experimental results confirmed fundamental concepts of computer architecture:

* **Cold misses** are inevitable and consistent, determined by the number of unique blocks accessed for the first time.
* **Conflict misses** dominate in small, direct-mapped caches and can be mitigated by increasing associativity.
* **LRU** consistently performed better than **FIFO** or **Random**, especially when the cache had higher associativity.

The simulator provided valuable insights and visual analytics, helping to connect theoretical knowledge with practical outcomes. It also demonstrated the trade-offs and design decisions that system architects must consider when optimizing memory hierarchies.

In conclusion, this project successfully met its objectives by building a working and flexible simulation tool, analyzing performance through experiments, and producing meaningful observations regarding cache behavior. This simulator can serve as a foundational tool for future extensions, such as supporting multi-level caches or exploring write policies.