Configurable Cache Memory Simulator in C

Course: CS-313 Computer Architecture

Ahmed Yar
Department of Computer Science
National University of Science and Technology (NUST)

***Abstract*—The significant and growing performance gap between modern CPUs and main memory, often called the "memory wall," necessitates the use of complex cache hierarchies. Designing and evaluating the effectiveness of these caches is a fundamental task in computer architecture. This project presents a flexible, event-driven cache simulator written in C. The simulator is designed to analyze cache performance by processing memory access traces. It supports configurable parameters for cache size, block size, and associativity (direct mapped, N-way set-associative, and fully associative). Furthermore, it implements two common replacement policies: Least Recently Used (LRU) and First-In, First-Out (FIFO). The program takes user-defined parameters and a memory trace file as input and outputs key performance metrics, including total hits, misses, and the overall hit rate, providing a tool for quantitative analysis of cache designs.**

***Index Terms*—Cache Memory, Simulator, Computer Architecture, LRU, FIFO, Associativity.**

I. Introduction

A. Background

In modern computing systems, processor speeds have increased at a much faster rate than main memory (DRAM) speeds. This disparity results in a performance bottleneck where the CPU frequently idles while waiting for data to be fetched from memory. **Cache memory** is a small, fast static RAM (SRAM) that sits between the CPU and main memory to bridge this gap. It stores frequently accessed data and instructions, allowing the CPU to access them much more quickly [1].

B. Problem Statement

The performance of a cache is not universal; it is highly dependent on its configuration and the nature of the program's memory access patterns. Key design choices include:

- **Cache Size:** A larger cache can hold more data but is slower and more expensive.
- **Associativity:** The flexibility of data placement, ranging from rigid direct mapping to fully flexible associative mapping.
- **Replacement Policy:** The algorithm used to decide which data to evict when the cache is full.

Manually calculating the performance of these complex interactions is unfeasible. Therefore, a simulator is an essential tool for architects to model and understand the trade-offs between different cache designs.

This work was performed as part of the requirements for the course **CS 313: Computer Architecture** at the National University of Science and Technology (NUST).

C. Project Objectives

The primary objectives of this project were to:

- Develop a modular and extensible cache simulator in the C programming language.
- Implement the three primary cache mapping organizations: **Direct-mapped**, **N-way Set-associative**, and **Fully Associative**.
- Implement two standard replacement policies: **LRU** and **FIFO**.
- Process external memory trace files to simulate realistic workloads.
- Calculate and report key performance metrics to evaluate the effectiveness of a given cache configuration.

D. Scope

This simulator models a single-level cache for **read accesses only**. It does not implement write policies (such as write-through or write-back) or multi-level cache hierarchies. The focus is purely on the performance implications of cache organization and replacement policies on memory read operations.

II. Theoretical Background

A. Cache Memory Principles

A cache is divided into several **cache lines** (or blocks), which are the fixed-size units of data transfer between the cache and main memory.

- **Cache Hit:** The requested data is found in the cache. This is a fast operation.
- **Cache Miss:** The requested data is not in the cache. The controller must fetch the block from main memory, incurring a "miss penalty."

B. Address Mapping

To determine where a memory block resides in the cache, its physical address is logically divided into three fields [1], [2].

- **Block Offset:** The least significant bits identify the specific byte within a cache block.
- **Index:** These bits determine the set where the memory block should be placed.
- **Tag:** The most significant bits are stored in the cache line to verify the block's identity.

C. Cache Organizations

- **Direct Mapped:** Each memory block maps to exactly one specific cache line. Pros: Simple hardware. Cons: High conflict misses.
- **N-Way Set-Associative:** The cache is divided into sets. A block maps to a set but can be placed in any of the N lines within that set. Pros: Reduces conflict misses. Cons: More complex hardware.
- **Fully Associative:** A memory block can be placed in any available cache line (single set). Pros: Lowest miss rate. Cons: Expensive hardware (Content-Addressable Memory).

D. Replacement Policies

When a set is full (on a cache miss), a policy decides which line to evict:

- **LRU (Least Recently Used):** Evicts the line accessed furthest in the past, relying on temporal locality.
- **FIFO (First-In, First-Out):** Evicts the line that has been in the cache the longest.

III. System Design and Implementation

A. Project Structure

The project follows a modular structure:

- `main.c`: Driver code and argument parsing.
- `cache.c/h`: Core simulation logic and struct definitions.
- `utils.c/h`: Bitwise operations and helper functions.

B. Core Data Structures

The simulation is built around two primary C structures:

1) CacheLine: Represents a single line within a set.

```
typedef struct {
    bool valid;
    uint64_t tag;
    uint32_t last_access_time; // For LRU
    uint32_t arrival_time;     // For FIFO
} CacheLine;
```

2) Cache: Represents the entire cache configuration and state.

```
typedef struct {
    CacheLine* lines; // Array of all cache lines
    CacheConfig config;
    CacheStats stats;
    uint32_t current_time; // Global logical clock
} Cache;
```

C. Cache Access Algorithm

The cache access function implements the core simulation logic:

1) **Clock Update:** Increment the global logical clock (current time).
2) **Parsing:** Extract **Tag** and **Index** from the memory address.
3) **Set Location:** Calculate the starting location of the relevant set in the lines array.
4) **Hit Check:** Iterate through the set. If (valid $\wedge$ tag match):
   - Increment Hit counter.
   - Update last access time (if LRU).
5) **Miss (Empty Line):** If no hit, search for an invalid (valid = false) line. If found, insert the new block (Tag) and update timestamps.
6) **Miss (Eviction):** If the set is full, call `find_victim()` (using LRU or FIFO logic) and overwrite the victim line.

IV. Testing and Methodology

A. Compilation and Execution

The project is compiled using a standard Makefile. The executable is run via Command Line Interface (CLI) with five arguments:

```
./cache_simulator <size> <block_size>
    <assoc> <policy> <trace.txt>
```

Example:

```
./cache_simulator 1024 64 4 FIFO trace.txt
```

B. Test Data

Validation was performed using `data/trace.txt`, a manually designed file containing hexadecimal addresses targeting the three types of cache misses: **Compulsory**, **Capacity**, and **Conflict**.

V. Results and Analysis

A. Simulation Output

Table I summarizes the output of a simulation run with the configuration: **1024-byte cache, 64-byte blocks, 1-way associativity (Direct Mapped), LRU Policy**.

TABLE I
Simulation Results

Metric	Value
Total Accesses	14
Total Hits	4
Total Misses	10
Hit Rate	28.57%
Miss Rate	71.43%

B. Discussion

The observed hit rate of 28.57% indicates that the **Direct mapped** configuration was suboptimal for the specific access pattern in the trace file, likely due to a high number of **conflict misses**. The simulator provides the capability to quantitatively verify architectural trade-offs; for instance, by increasing the associativity (e.g., to 4-way) in subsequent tests, one can observe the corresponding reduction in conflict misses and the resulting improvement in the overall hit rate.

VI. Conclusion and Future Work

A. Conclusion

This project successfully achieved its objectives by creating a functional, configurable cache simulator in C. The tool correctly implements multiple cache organizations and replacement policies, serving as an effective platform for analyzing architectural trade-offs in computer systems.

B. Future Work

Potential extensions to enhance the simulator's capability include:

- **Write Policies:** Implementation of Write-Through and Write-Back policies using "dirty bits."
- **Multi-Level Hierarchies:** Extension to model L1/L2 cache configurations and their interaction.
- **Advanced Policies:** Implementation of less common but viable replacement policies like Random or Least Frequently Used (LFU).

Acknowledgment

The author would like to thank the faculty of the Department of Computer Science at NUST for their guidance and support throughout this project.

References

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th ed. Morgan Kaufmann, 2017.

[2] J. L. Hennessy and D. L. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.