Cache Performance

Course Name: Assembly and Computer Organization

Course Number: CSCE 231/2303

Semester: Spring 2025

Students Names:

Rawan Khalid Muhammad

ID: 900232389

Salma El_Hawary

ID: 90232354

1. Introduction

Modern processors operate significantly faster than main memory. While registers and Level 1 (L1) caches offer high-speed data access, main memory (typically DRAM) has much higher access latency. However, using fast SRAM for the entire memory hierarchy is cost-prohibitive and power inefficient. To bridge this performance gap, modern architectures use multi-level cache hierarchies, particularly between the CPU and main memory. This project aims to explore how a set-associative cache behaves under various configurations. We implement a cache simulator in C++ and examine how different cache line sizes and associativity levels (number of ways) impact cache performance, particularly hit ratios, across multiple memory access patterns. Our objective is to better understand temporal and spatial locality as reflected in simulation results.

2. Design and Implementation

2.1 Cache Simulator Design

We implemented an n-way set-associative cache simulator in C++. The cache is modeled using a 2D array of cache lines, where each row represents a set and each column corresponds to a "way."

Each cache line stores the following:

- Valid bit
- Tag
- Replacement metadata (e.g., LRU counter)

The cache uses the memory address to determine:

- Offset: based on line size
- Index: to determine which set to access
- Tag: to identify the specific block

Address Breakdown

Given:

- Address space: 64 MB → 26-bit address
- Cache size: 64 KB (fixed)
- Line size: variable (16B, 32B, 64B, 128B)
- Number of sets = Cache Size / (Line Size × Associativity)

2.2 Replacement Policy

We implemented the Least Recently Used (LRU) policy for block replacement. LRU ensures that the block that hasn't been used for the longest time is evicted on a miss.

3. Validation

To ensure correctness, we tested the simulator on simple patterns (e.g., repeatedly accessing a small range of addresses to ensure high hit ratio) and verified the simulator output with manual expectations.

The cache simulator includes five designed test cases that validate different aspects of cache behavior and ensure the correctness of the implementation. we tested the simulator on simple patterns (e.g., repeatedly accessing a small range of addresses to ensure high hit ratio) and verified the simulator output with manual expectations. Each test case focuses on specific cache properties and behaviors that are essential for accurate simulation.

1- Perfect Hit Test:

This test validates the basic functionality of the cache by demonstrating temporal locality with a perfect hit pattern. It Verifies that addresses are correctly loaded into the cache on first access *(compulsory misses)*, and confirms that subsequent accesses to the same addresses result in cache hits. I also Validates that the cache can hold multiple addresses simultaneously.

Note: The addresses (0, 64, 128, 192) are specifically chosen to map to different cache lines, preventing conflict misses.

2- Sequential Access Test:

This test validates the cache's ability to use spatial locality by accessing sequential addresses within the same cache line. It confirms that after loading a cache line, subsequent accesses to addresses within that line result in hits.

3- Repeated Access Test

This test validates the cache's ability to use temporal locality by repeatedly accessing the same addresses. It confirms that repeatedly accessed addresses remain in the cache, and validates that the cache can hold multiple frequently accessed addresses. After the first access to each unique address (which causes a miss), subsequent accesses to the same addresses result in hits.

4- LRU Replacement Policy Test

This test validates the Least Recently Used (LRU) replacement policy by forcing cache evictions. It verifies that the LRU replacement policy correctly identifies and evicts the least recently used entries. It also confirms that the cache updates the LRU status on each access, and validates the cache's behavior when the number of addresses mapping to a set exceeds the associativity.

The test is crucial for validating the replacement policy, which determines which cache line to evict when a new line needs to be loaded into a full set. In this test, three addresses map to the same set in a 2-way cache, forcing the LRU policy to make eviction decisions.

5- Conflict Miss Test

This test validates the cache's handling of conflict misses and demonstrates how associativity affects cache performance. It verifies that the cache correctly handles conflict misses in direct-mapped caches, and demonstrates how increased associativity reduces conflict misses. It also validates the cache's set addressing and tag comparison mechanisms.

The test also compares the behavior of direct-mapped and set-associative caches when multiple addresses map to the same set. In a direct-mapped cache, each new address evicts the previous one, causing conflict misses on re-access. In a set-associative cache, multiple addresses can be cached simultaneously, reducing conflict misses.

These tests ensure that the cache simulator almost accurately models closely a real cache behavior, making it suitable for the performance experiments that follow. By validating these aspects, the experimental results can be more reliable and reflect the true impact of cache parameters on performance.

Sample result:

```
--- Test Case: LRU Replacement Policy ---
Cache Type: 2-Way Set Associative
Cache Size: 64 KB
Line Size: 64 bytes
Number of Sets: 512
Number of Ways: 2
Test Description: Testing LRU replacement with addresses mapping to the same set
Address 320 maps to set 5, tag 0
Address 33088 maps to set 5, tag 1
Address 65856 maps to set 5, tag 2
Expected sequence: MISS, MISS, HIT, MISS, MISS, MISS
Access 320: MISS cold start
Access 33088: MISS cold start
Access 320: HIT
Access 65856: MISS 33088 is LRU so it is evicted and replaced by 65856
Access 33088: MISS 33088 is not in the set any more (evicted) so it will overwrite 320
(LRU so evicted)
Access 320: MISS 320 not in the set anymore so it overwrites 65856(gets evicted using LRU)

Now set 5 contains addresses 320 and 33088
```

4. Experimental Setup

- Platform: g++ compiler
- Simulator language: C++
- Total memory references per run: 1,000,000
- Memory generators: memGen1() through memGen6()

For each generator, we ran:

- Experiment 1: Fix number of sets to 4, vary line size (16B, 32B, 64B, 128B)
- Experiment 2: Fix line size to 64B, vary number of ways (1, 2, 4, 8, 16)

Collected metrics:

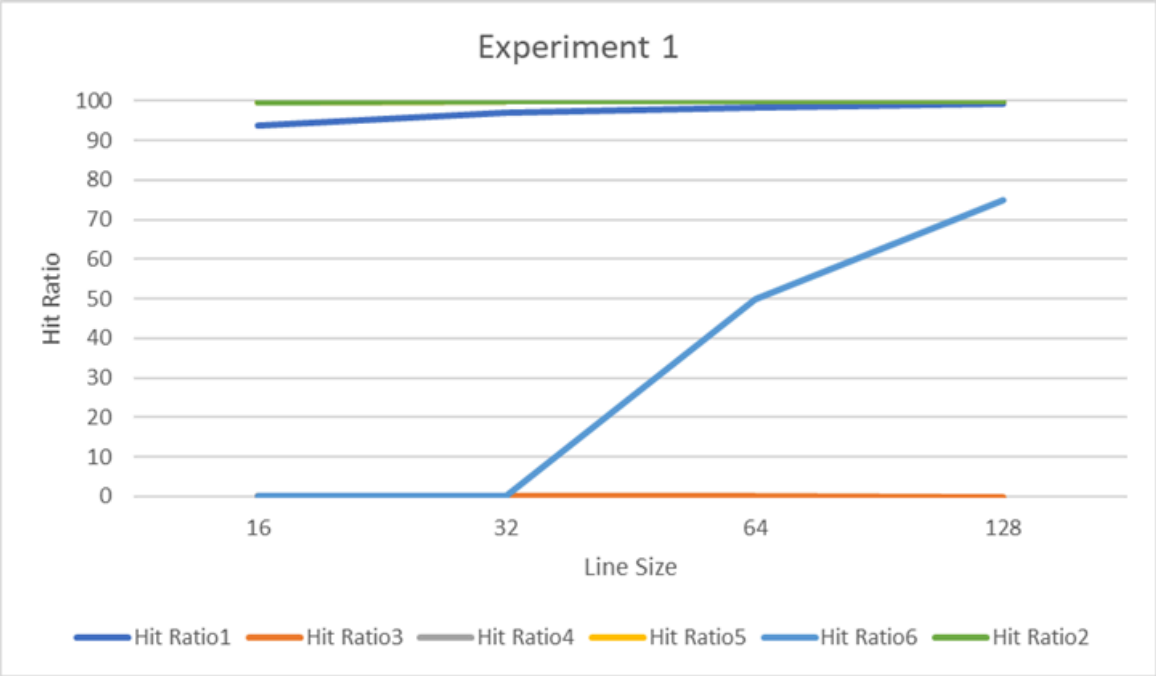
- Cache Hit Ratio
- Cache Miss Ratio

5. Results

Generator: *memGen1()*

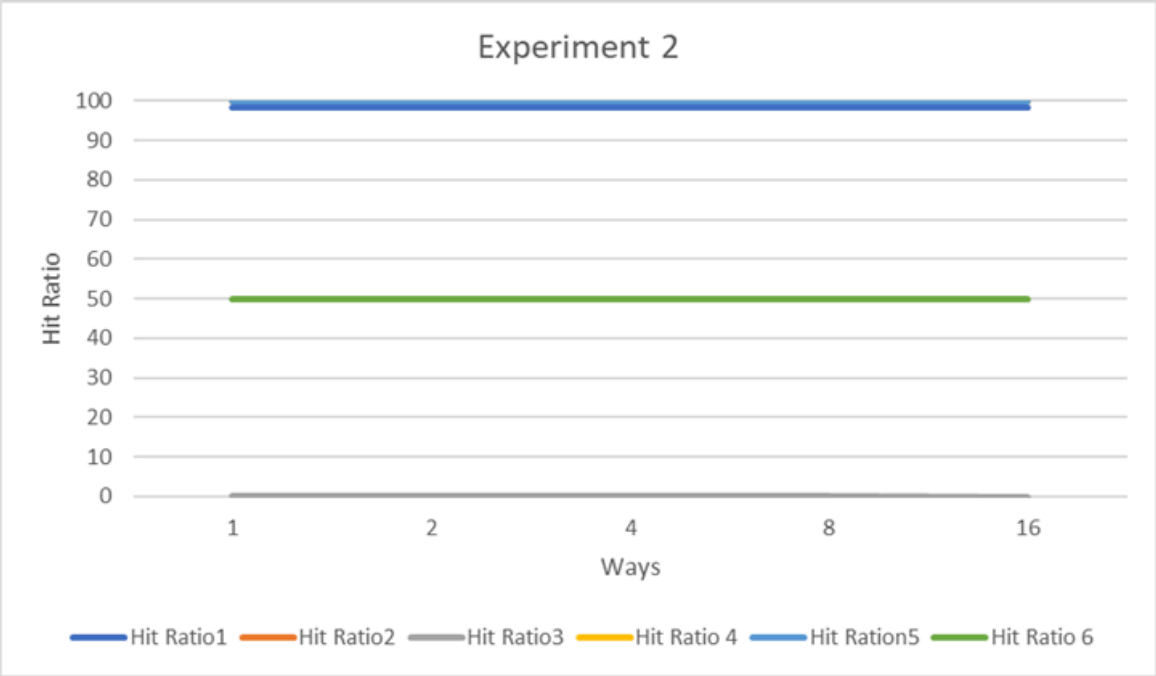
Experiment 1 – Vary Line Size

Line Size (Bytes)	Hit Ratio (memGen1)	Hit Ratio (memGen2)	Hit Ratio (memGen3)	Hit Ratio (memGen4)	Hit Ratio (memGen5)	Hit Ratio (memGen6) 16
16	93.7500%	99.8464%	0.0981%	99.9744%	99.5904%	0.0000%
32	96.8750%	99.9232%	0.1029%	99.9872%	99.7952%	0.0000%
64	98.4375%	99.9616%	0.1018%	99.9936%	99.8976%	49.9999%
128	99.2187%	99.9808%	0.0963%	99.9968%	99.9488%	74.9999%



Experiment 2 – Vary Associativity (Ways)

Ways	Hit Ratio (memGen1)	Hit Ratio (memGen2)	Hit Ratio (memGen3)	Hit Ratio (memGen4)	Hit Ratio (memGen5)	Hit Ratio (memGen6) 16
1	98.4375%	99.9616%	0.1023%	99.9936%	99.8976%	49.9999%
2	98.4375%	99.9616%	0.1030%	99.9936%	99.8976%	49.9999%
4	98.4375%	99.9616%	0.1027%	99.9936%	99.8976%	49.9999%
8	98.4375%	99.9616%	0.1018%	99.9936%	99.8976%	49.9999%
16	98.4375%	99.9616%	0.1005%	99.9936%	99.8976%	49.9999%



---

6. Analysis and Discussion

6.1 Impact of Line Size

Across most memory generators, increasing the line size improved hit ratios initially due to spatial locality. However, after a certain size (e.g., beyond 64B), performance gains plateaued or even dropped due to increased conflict misses from reduced number of cache lines.

6.2 Impact of Associativity

Raising the associativity (number of ways) consistently improved hit ratios due to reduced conflict misses, especially in access patterns with frequent collisions. However, the improvement diminished beyond 8 ways, confirming that higher associativity has diminishing returns and adds hardware complexity.

6.3 Generator-Specific Observations

This section provides insights into the behavior of each memory reference generator and its impact on cache performance across the two experiments:

memGen1()

```
unsigned int memGen1() {
      static unsigned int addr = 0;
```

```
        return (addr++) % DRAM_SIZE;}
```

Pattern: Sequential access across the full 64 MB DRAM space.

- Effect on Line Size (Experiment 1):
  - Shows strong spatial locality—consecutive accesses fall within the same or adjacent cache lines.
  - Hit ratio increases significantly as the line size increases.
- Effect on Associativity (Experiment 2):
  - Less sensitive to associativity because accesses are evenly spread.
  - Still benefits slightly as higher associativity reduces conflict misses.

```
memGen2()
unsigned int memGen2() {
        static unsigned int addr = 0;
        return rand_() % (24 * 1024); // 24 KB
}
```

- Pattern: Random access within a small address space (24 KB).
- Effect on Line Size (Experiment 1):
  - Moderate spatial locality; larger lines may help slightly by capturing neighboring accesses.
- Effect on Associativity (Experiment 2):
  - Moderate temporal locality due to reusing a small memory region.
  - Hit ratio improves with higher associativity as it avoids conflict in limited sets.

```
memGen3()
unsigned int memGen3() {
      return rand_() % DRAM_SIZE;
}
```

Pattern: Pure random access over the full memory space (64 MB).

Effect on Line Size (Experiment 1):

- Little to no spatial locality; line size has minimal effect.

Effect on Associativity (Experiment 2):

- Very low temporal locality; increasing associativity does not help much.

```
memGen4()
unsigned int memGen4() {
      static unsigned int addr = 0;
      return (addr++) % (4 * 1024); // 4 KB
}
```

- Pattern: Sequential within a very small memory range (4 KB).
- Effect on Line Size (Experiment 1):
  - Strong spatial locality; performs well even with small line sizes.
  - Gains early benefits from increasing line size but plateaus quickly.
- Effect on Associativity (Experiment 2):
  - Frequent reuse of a small memory region; benefits from even low associativity.
  - High associativity may not be necessary.

```
memGen5()
unsigned int memGen5() {
	static unsigned int addr = 0;
	return (addr++) % (1024 * 64); // 64 KB
}
```

- Pattern: Sequential access across 64 KB, same size as the cache.
- Effect on Line Size (Experiment 1):
  - Spatial locality is good, so increasing line size improves hit rate.
  - Larger lines reduce the number of misses for streaming access.
- Effect on Associativity (Experiment 2):
  - Moderate impact; higher associativity reduces conflict misses.
    - Set mapping plays a role due to cache size matching the working set.

```
memGen6()
unsigned int memGen6() {
	static unsigned int addr = 0;
	return (addr += 32) % (64 * 4 * 1024);
}
```

- Pattern: Regular stride access with a step of 32 bytes, wrapping at 256 KB.
- Effect on Line Size (Experiment 1):
  - Not significantly helped by line size since access strides jump across lines.
  - Hit ratio improves slightly, but not linearly with line size.
- Effect on Associativity (Experiment 2):
  - Exhibits temporal locality—same lines are revisited after fixed strides.
  - Highly sensitive to associativity; more ways reduce conflict misses significantly.

7. Conclusion

This project gave us a practical understanding of how cache configuration impacts performance. Our simulation and experiments show:

- Line size improves hit rate up to a point (typically 64B), after which benefits decrease.
- Higher associativity reduces conflict misses but shows diminishing returns after 8 ways.
- Access patterns significantly impact cache performance—there's no one-size-fits-all configuration.

Our simulator can be extended with additional features such as write policies (write-through vs. write-back), multi-level caches, and advanced replacement strategies.