COL216 Assignment-3 Report

Aditya Garg (2023CS10235)
Vatsal Malav (2023CS10430)

April 30, 2025

1 Introduction

This report analyzes an L1 cache simulator implementation designed for multicore processors using the MESI cache coherence protocol. The simulator models a system with four cores, each with its own L1 cache, connected via a central bus. The system employs a write-back, write-allocate write policy and uses LRU (Least Recently Used) for cache line replacement.

2 Key Classes and Data Structures

2.1 CacheLine and CacheSet

The `CacheLine` structure represents a single cache line with:

- **valid** flag to indicate if the line contains valid data
- **dirty** flag to indicate if the line has been modified
- **state** to store the MESI protocol state
- **tag** to store the address tag

The `CacheSet` structure contains multiple cache lines based on the associativity parameter.

2.2 Cache Class

The `Cache` class maintains:

- Configuration parameters (s, E, b)
- Statistics counters for cache performance
- Vector of cache sets
- Methods for address decomposition
- Methods for cache access and snooping

2.3 BusManager Class

The `BusManager` class implements the bus interface and coherence protocol:

- Keeps track of all caches in the system
- Manages bus arbitration and timing
- Implements bus operations (BusRd, BusRdX, Invalidate)
- Tracks bus statistics

2.4 Event-driven Simulation Data Structures

- `Operation` represents a single memory operation
- `Event` represents a scheduled cache operation with timing

3 Key Functions and algorithms

3.1 Cache Access Function

The `Cache::access` function handles read (`R`) and write (`W`) operations requested by the core. Its main tasks are:

- **Tag and Index Extraction:** The cache index and tag are extracted from the address.
- **Cache Hit:**
  - For reads, the LRU is updated; no state change is needed.
  - For writes:
    - If the line is `SHARED`, a bus invalidate is broadcasted to move it to `MODIFIED`.
    - If `EXCLUSIVE`, it silently upgrades to `MODIFIED`.
- **Cache Miss:**
  - A victim line is chosen and, if dirty, is written back to memory.
  - A `BusRd` (for reads) or `BusRdX` (for writes) is broadcasted.
  - The line's new state is set based on bus responses: `SHARED`, `EXCLUSIVE`, or `MODIFIED`.
- **Cycle Accounting:** Total cycles, idle cycles, and traffic are updated based on memory and bus operations.

3.2 Cache Snoop Function

The `Cache::snoop` function responds to bus operations from other cores:

- **BusRd:**
  - `EXCLUSIVE` lines move to `SHARED` and supply data.
  - `MODIFIED` lines write back to memory before sharing.
- **BusRdX:**
  - Lines in `SHARED`, `EXCLUSIVE`, or `MODIFIED` are invalidated.
  - `MODIFIED` lines also perform a memory writeback.
- **Invalidate:**
  - `SHARED` lines are invalidated.
- **Traffic and Counters:** State transitions, writebacks, and data traffic counters are updated.

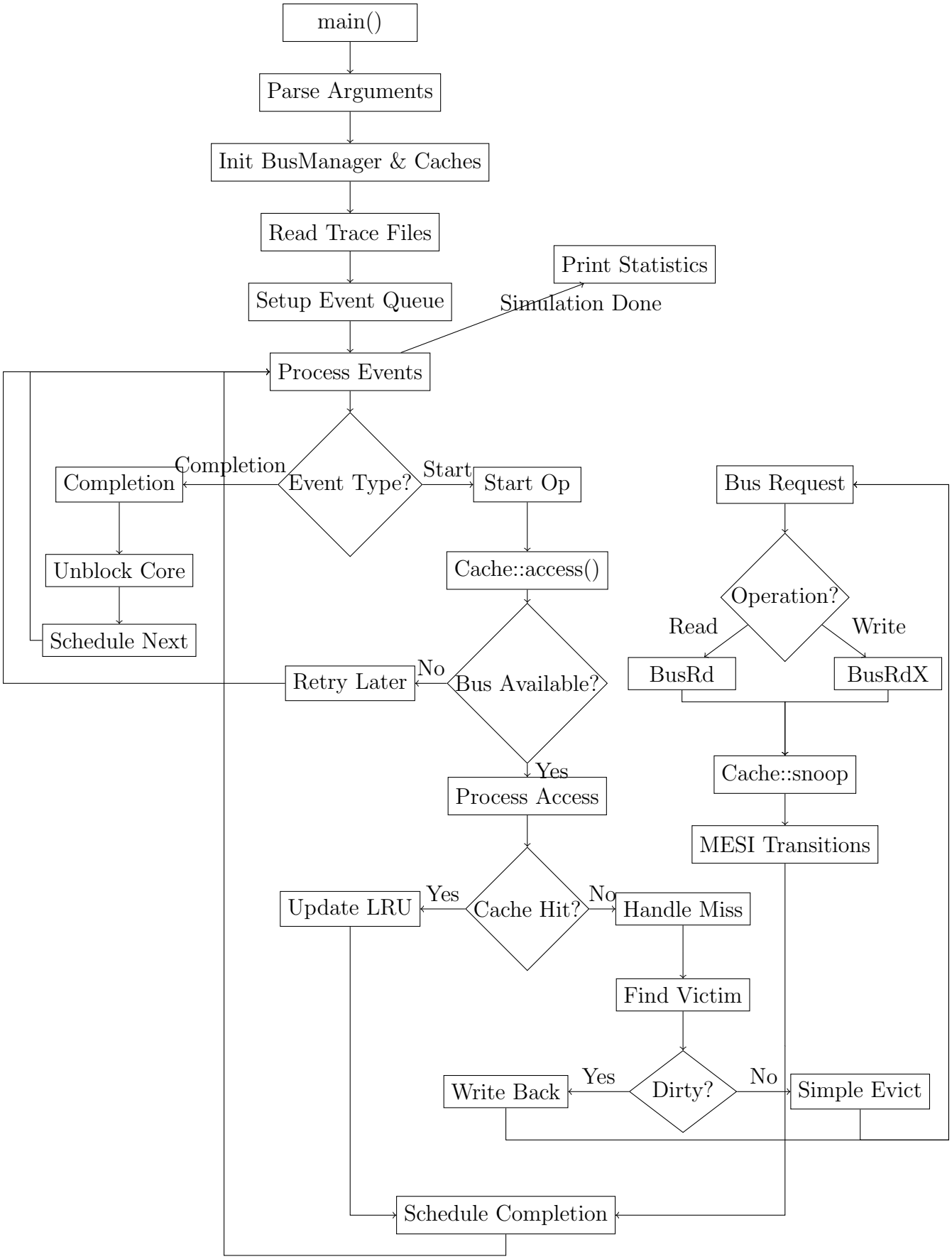
3.3 Bus Contention Handling

If the bus is busy, the access is retried and `idle_cycles` is incremented to account for waiting.

3.4 Main Function Overview

The `main` function orchestrates the simulation as follows:

- **Parse Arguments:** Reads trace prefix, cache parameters (`s`, `E`, `b`), and optional output file. Displays help on invalid input.
- **Setup Output:** Results are printed to console or written to a file if specified.
- **Initialize Simulation:**
  - Creates four cache instances and registers them with a central `BusManager`.
- **Load Trace Files:**
  - Preprocesses each core's trace file, storing memory operations.
- **Run Event-Driven Simulation:**
  - Schedules and executes memory operations per core using a priority queue.
  - Handles bus contention, cache hits/misses, and operation blocking.
- **Report Results:**
  - Prints core statistics, bus transactions, total traffic, execution time, and cache utilization.
- **Cleanup:** Deletes cache objects and closes the output file if necessary.


main()
Parse Arguments
Init BusManager & Caches
Read Trace Files
Setup Event Queue
Print Statistics
Simulation Done
Process Events
Event Type?
Completion
Start
Completion
Start Op
Unblock Core
Schedule Next
Cache::access()
Bus Available?
No
Retry Later
Yes
Process Access
Cache Hit?
Yes
Update LRU
No
Handle Miss
Find Victim
Dirty?
Yes
Write Back
No
Simple Evict
Schedule Completion
Bus Request
Operation?
Read
Write
BusRd
BusRdX
Cache::snoop
MESI Transitions

4 Assumptions

- The simulator models an L1 cache per core, using the MESI coherence protocol and a central snooping bus.
- All signals and data transfers require access to the bus. If the bus is busy, cores must wait.
- Cache hits and misses increment execution cycles by 1.
- Cache misses may lead to evictions. Dirty evictions cost 100 execution cycles for the evicting core.
- Memory write operations (e.g., due to write misses) take 100 cycles, but cores do not stall; they drop the line on the bus and continue.
- In cache-to-cache transfers, the responding core incurs $2N$ execution cycles (where N is the number of words transfered).
- Waiting for bus access is counted as idle cycles.
- All coherence signals (e.g., `BusRd`, `BusRdX`, `Invalidate`) are sent instantly and take effect immediately.
- If an `Invalidate` is sent but no core holds the target block, the invalidation counter is still incremented.
- Lower core IDs are prioritized when multiple cores contend for the bus in the same cycle.
- Cores are not stalled during dirty block writebacks to memory or while responding to cache-to-cache transfers. However, cores are stalled if they are waiting for the bus to become free or waiting for the data to arrive.
- Data traffic includes all data sent or received by a core: cache-to-cache transfers, memory read responses, and writebacks to memory. In the case of a read miss, if a responding core supplies the data (cache-to-cache transfer) and also performs a writeback to memory, both transfers are counted toward total data traffic.
- Bus transactions include memory fetches, cache-to-cache transfers, and invalidation broadcasts. Simple reads or writes that hit in the cache do not use the bus. Therefore, each time a `BusInvalidate`, `BusRd`, or `BusRdX` signal is sent on the bus, the bus transactions counter is incremented.

  Note: This simulator currently uses the definition of execution cycles as described in this section — counting only the cycles spent actively executing instructions, including those involving cache hits, misses, and coherence protocol effects. Idle cycles, such as waiting for the bus to be free or waiting for data to arrive, are tracked separately. However, if we instead use the definition: "Execution cycles count cycles spent executing its own instruction traces, and idle cycles are cycles spent waiting for the resources while other cores are executing its instructions," and include such wait times within execution cycles, then `execution_cycles` in the `print_stats()` function should be replaced with `total_cycles` to match that interpretation.

5 Performance Analysis

5.1 Core Performance Comparison

The simulator was run with the following parameters: trace prefix `app1`, 6 set index bits (64 sets), associativity of 2, and block size of 32 bytes (5 block bits), resulting in a 4 KB cache per core. Figure 1 shows a comparison of key performance metrics across all four cores.

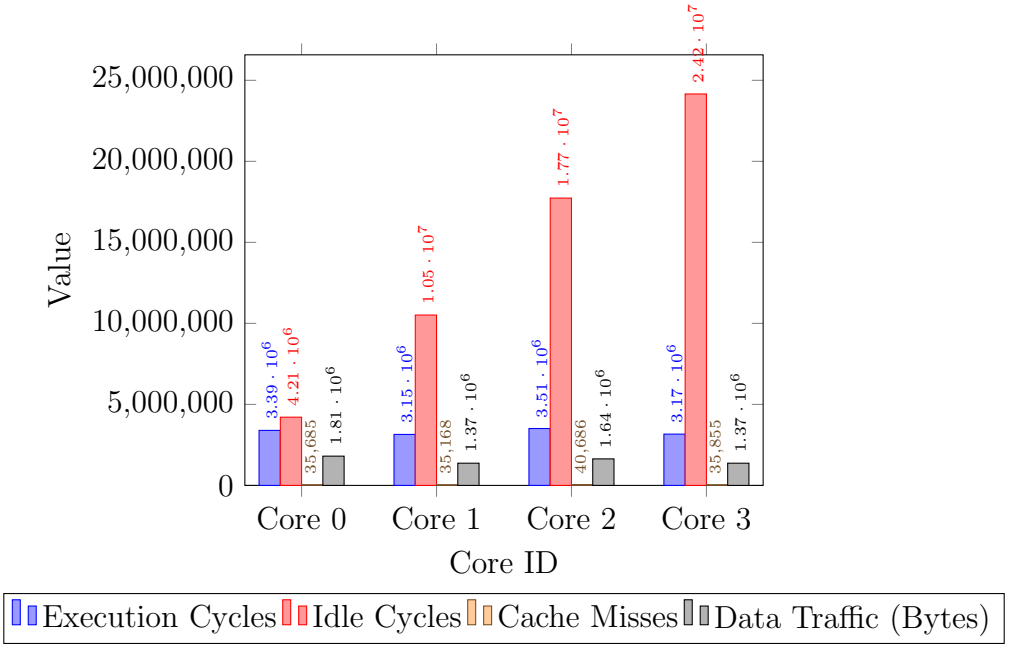


Figure 1: Performance comparison across the four cores

From the data, we observe that Core 2 has the highest execution time and cache miss rate, indicating it may be handling more complex or cache-intensive operations. The idle cycles increase progressively from Core 0 to Core 3, suggesting a sequential dependency or bus contention effects where higher-numbered cores wait longer for bus access, consistent with the prioritization of lower core IDs during bus contention. Despite similar instruction counts, execution time varies by up to 11% between cores, highlighting the impact of cache behavior and bus arbitration.

It's important to note that on multiple runs, these results will not change because core selection for bus arbitration is deterministic based on core ID. This ensures that all operations are performed in the same order across runs, making the simulation results reproducible.

5.2 Impact of Cache Parameters on Performance

To better understand how different cache parameters affect overall system performance, we conducted a series of experiments varying cache associativity, block size, and cache size while keeping other parameters constant. The following analyses present the maximum execution time across all cores for each configuration.

5.2.1 Cache Associativity

Figure 2 shows the relationship between cache associativity and maximum execution time.

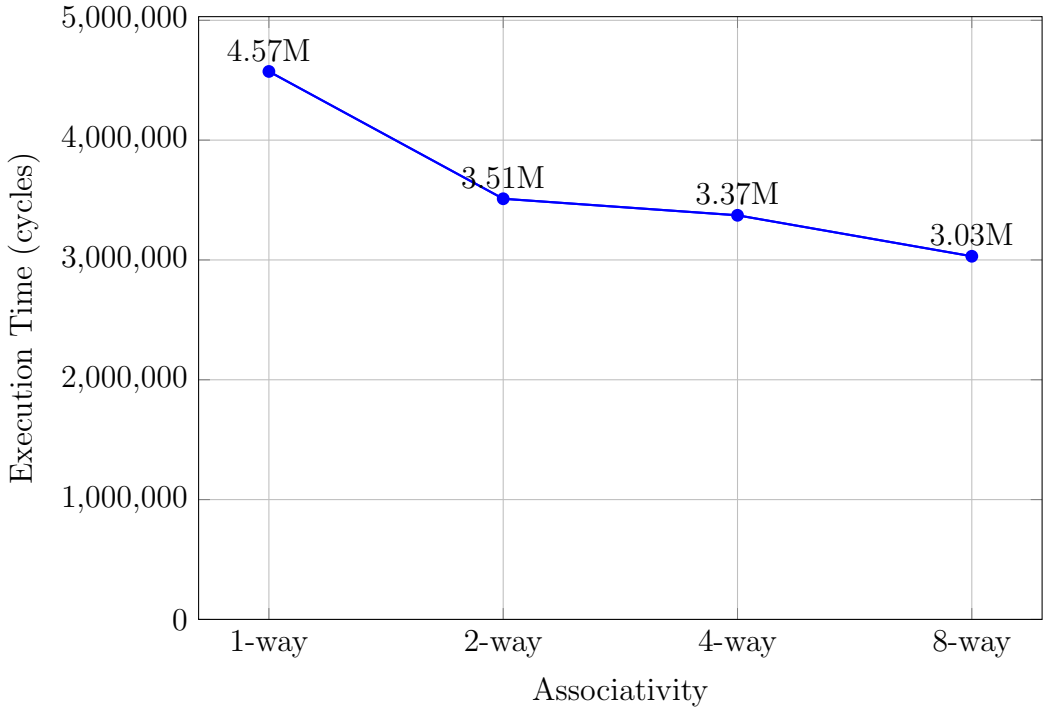


Figure 2: Cache Associativity vs. Execution Time

The results clearly show that increasing cache associativity leads to significant reductions in execution time. Moving from a direct-mapped cache (1-way) to a 2-way set-associative cache yielded a 23% improvement, while further increases to 4-way and 8-way provided additional gains of 4% and 10% respectively. Higher associativity reduces conflict misses by allowing multiple blocks that map to the same set to coexist in the cache, thus improving temporal locality exploitation. However, the diminishing returns suggest that for this workload, an 8-way associative cache provides a good balance between performance and implementation complexity.

5.2.2 Block Size

Figure 3 illustrates how block size affects execution time.

Interestingly, block size exhibits a non-monotonic relationship with execution time. Performance improves when moving from 16B to 32B blocks (8.7% reduction in execution time), but then significantly degrades with larger block sizes. This U-shaped curve indicates competing effects: while larger blocks can improve spatial locality by prefetching nearby data, they also increase transfer times, bus contention, and pollution when only a small portion of the block is used. For this workload, 32B appears to be the optimal block size, balancing spatial locality benefits with overhead costs. The dramatic performance drop with 128B blocks (127% slower than optimal) suggests severe cache pollution and increased data transfer overhead.

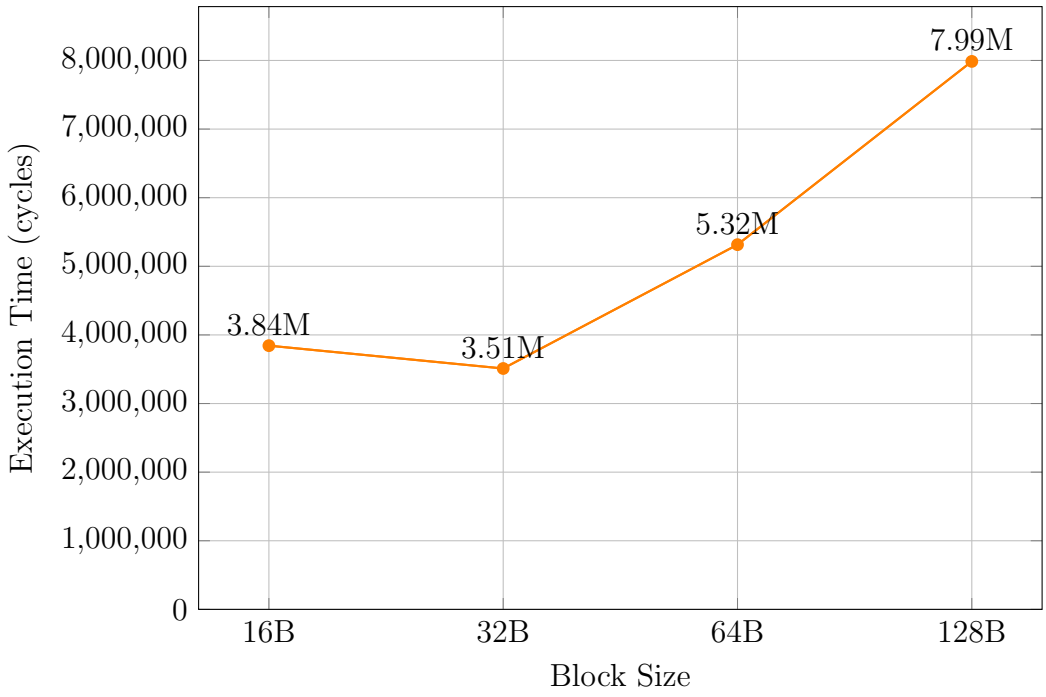


Figure 3: Block Size vs. Execution Time

5.2.3 Cache Size

Figure 4 shows the impact of cache size on execution time.

As expected, increasing cache size results in improved performance, with execution time decreasing as cache size increases. The most significant improvement occurs when doubling the cache from 4KB to 8KB (15.5% reduction), followed by more modest gains for further increases. This suggests that the working set of the application fits partially within 8KB, with diminishing returns beyond that size. The flattening curve between 16KB and 32KB (only 1.6% improvement) indicates that most of the working set fits within 16KB, and further increases in cache size provide minimal benefit for this workload.

5.3 Overall Findings

Our performance analysis reveals several important insights:

- For this workload, increasing associativity consistently improves performance, with 8-way providing the best results among tested configurations.
- Block size exhibits a more complex relationship with performance, with 32B being optimal. Larger blocks significantly harm performance due to increased transfer overhead and cache pollution.
- Cache size shows diminishing returns beyond 8KB, suggesting that the working set of this application fits mostly within this size.
- Core performance varies despite similar instruction counts, with Core 2 exhibiting higher miss rates and execution times.
- Idle cycles increase with core ID due to bus arbitration prioritization, with Core 3 experiencing the highest wait times.

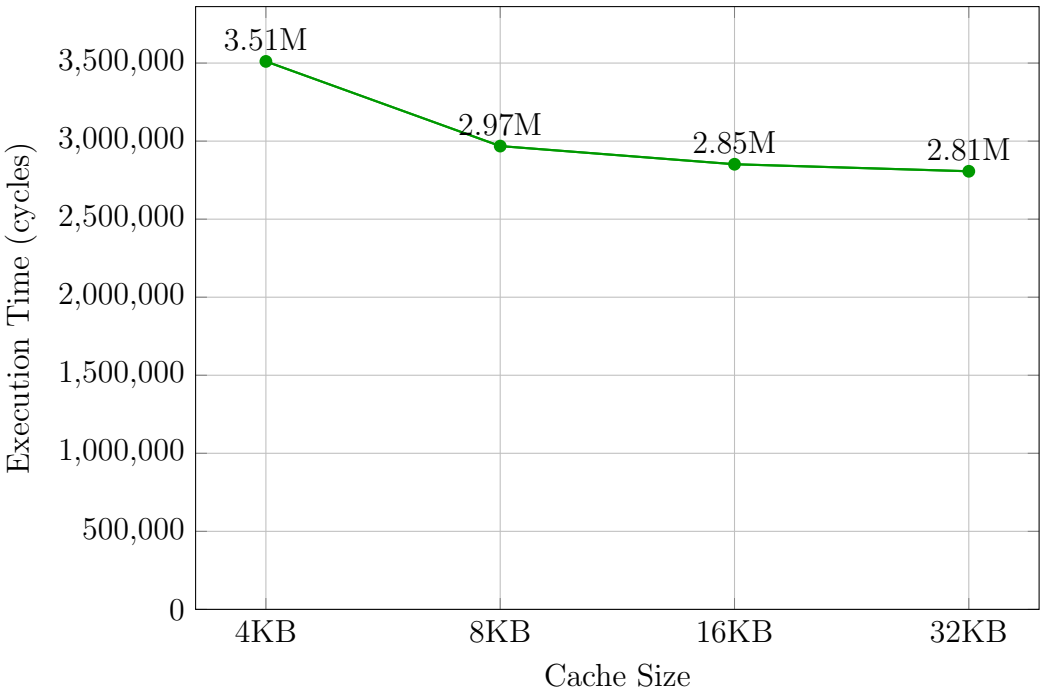


Figure 4: Cache Size vs. Execution Time

These findings highlight the importance of properly tuning cache parameters to match the characteristics of the target workload. The most significant performance gains for this application would come from selecting higher associativity, moderate block sizes (32B), and sufficient cache capacity (at least 8KB).

6 Conclusion

This report presented an L1 cache simulator for a multicore system using the MESI coherence protocol. The simulator effectively models cache operations, bus contention, and coherence events, while tracking key metrics like execution cycles, idle time, and data traffic.

Execution cycles currently exclude stalls from bus or data wait times. If a broader definition is preferred, these should be included by replacing `execution_cycles` with `total_cycles` in the stats. The simulator provides a flexible and efficient framework for analyzing coherence behavior and performance in multicore systems.