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.

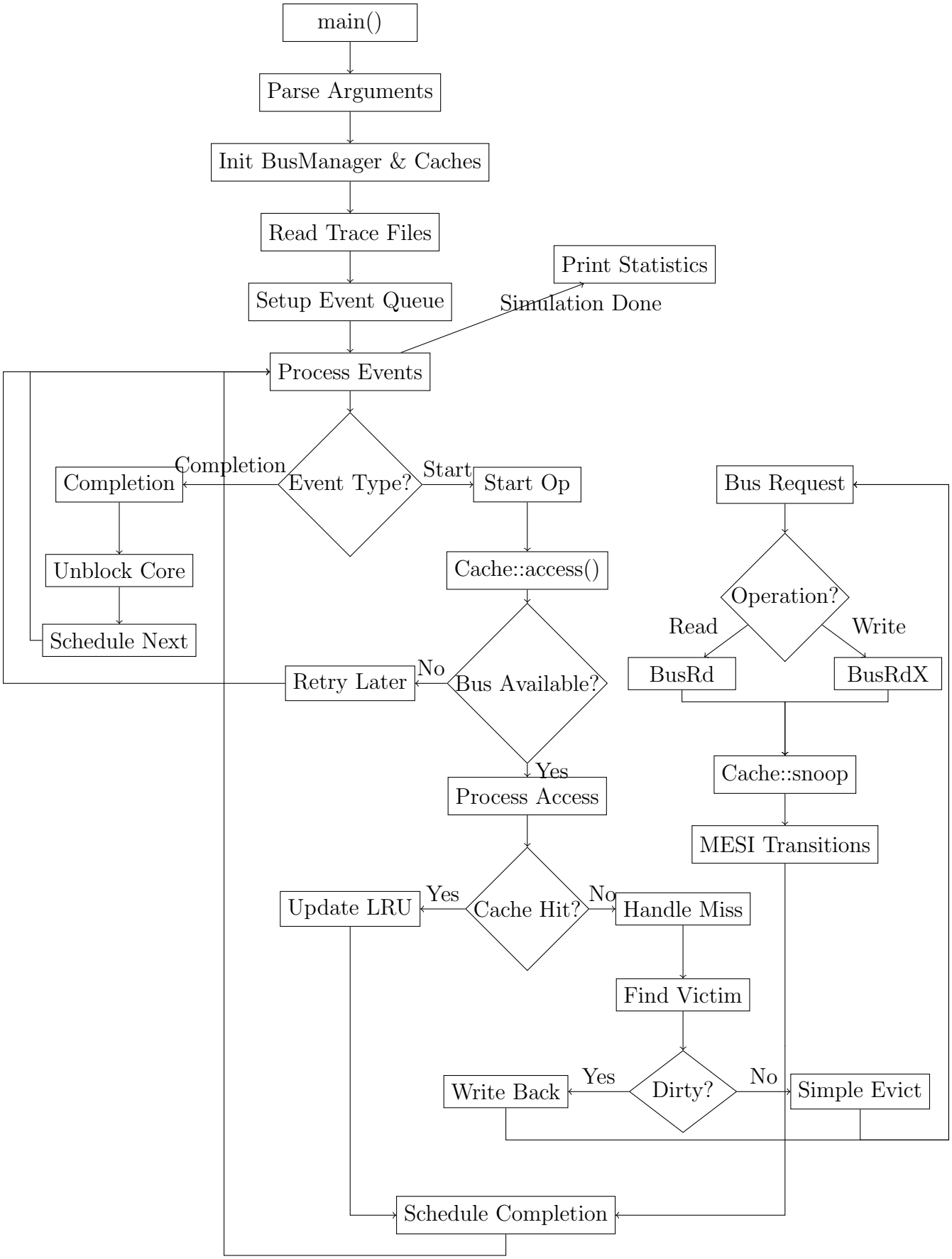
3.3 Bus Contention Handling

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.
- **Execution cycles:** These are all cycles during which a core is processing its own instructions. This includes waiting for data from memory, evicting a block, or waiting for a cache-to-cache transfer to complete.
- **Idle cycles:** These are cycles where a core is stalled solely because it is waiting for the bus to become free while other cores are executing their instructions. Waiting for data to arrive is not counted as idle time.
- 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 to complete, but cores do not stall; they initiate the write and continue execution.
- In cache-to-cache transfers, the responding core is **not stalled**, so no cycles are added to its execution or idle time.
- 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.
- 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.

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.

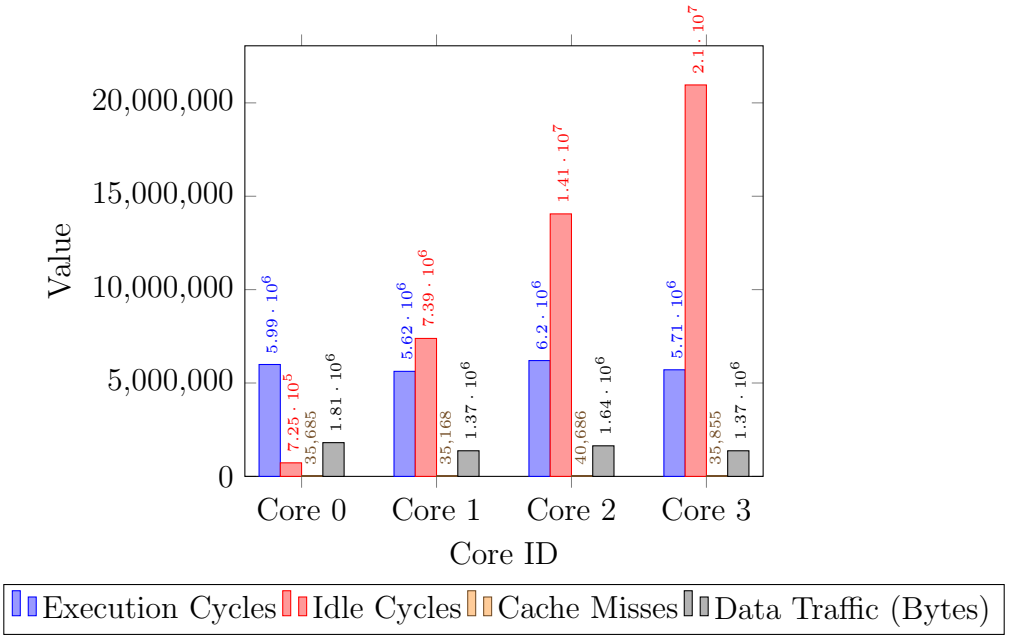


Figure 1: Performance comparison across the four cores

From the data, we observe that Core 3 has the highest idle cycles, followed by Cores 2, 1, and 0 in increasing order. This trend suggests significant bus contention or delayed access for higher-numbered cores, which is consistent with the design assumption where lower core IDs are prioritized during bus arbitration. In contrast, Core 0 exhibits the lowest idle time and moderate execution cycles, indicating more efficient access to the bus and cache. Core 2 has a noticeably higher execution cycle count, which, along with a relatively high cache miss and data traffic count, suggests it may be performing memory-intensive operations. The variation in execution and idle cycles across cores—even with similar data traffic—highlights the role of cache behavior, eviction delays, and bus contention in overall performance.

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

In this section, we explore the effect of various cache parameters on system performance, focusing on cache size, associativity, and block size. Each parameter was varied individually while keeping the others fixed to better understand their independent impact.

5.2.1 Cache Size

The results in Figure 2 show that as cache size increases, execution time consistently decreases. This trend indicates that larger caches reduce the number of cache misses, thereby decreasing the number of memory accesses. Notably, the most significant improvement occurs when increasing from 4KB to 8KB, which provides an approximate 16.7

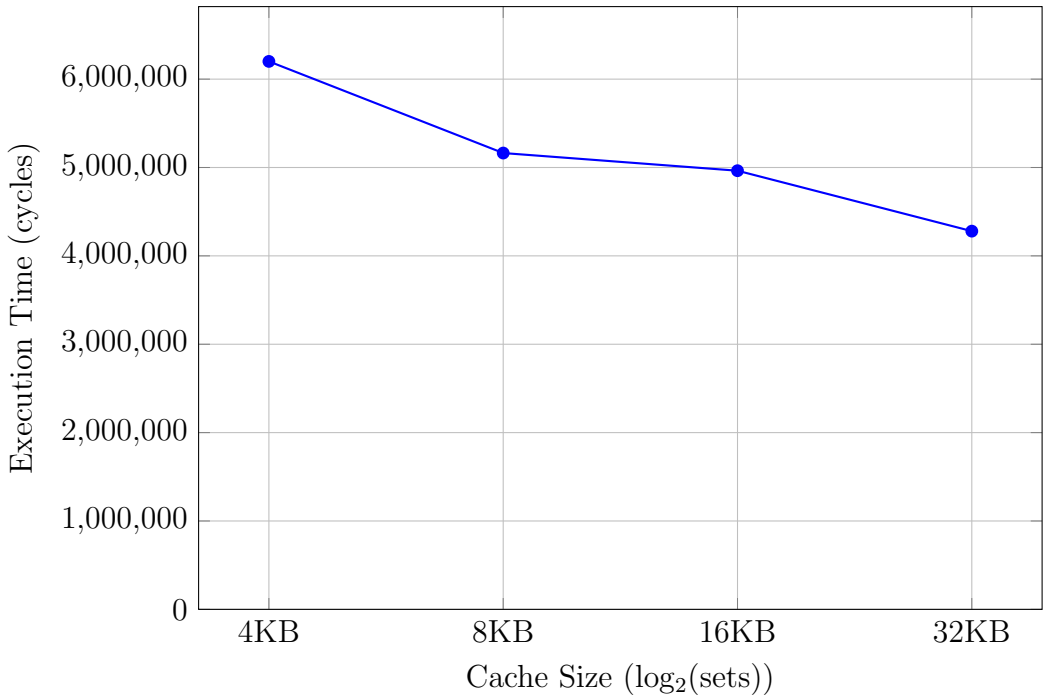


Figure 2: Cache Size vs. Execution Time

5.2.2 Cache Associativity

Cache associativity plays a significant role in reducing conflict misses, as shown in Figure 3. The direct-mapped (1-way) configuration experiences very high execution times due to excessive conflict misses. By increasing the associativity to 2-way, execution time drops by about 81

5.2.3 Block Size

As shown in Figure 4, the optimal block size for this workload is 32B. Initially, increasing the block size from 16B to 32B results in a significant reduction in execution time, likely due to better utilization of spatial locality in memory. However, further increases in block size (to 64B and 128B) cause a slight performance degradation. This is likely due to the increased bus traffic and cache pollution resulting from loading larger blocks that contain more data than required, which can cause cache evictions. The increased overhead from handling larger blocks and transferring unused data outweighs the benefits of increased spatial locality beyond 32B. This suggests that a block size of 32B is the sweet spot for this particular workload, where the benefits of spatial locality are maximized without incurring excessive overhead.

5.3 Overall Findings

Our performance analysis provides the following insights:

- Increasing associativity consistently improves performance. The 8-way associativity configuration offers the best performance, with higher associativity significantly reducing cache misses and execution time.

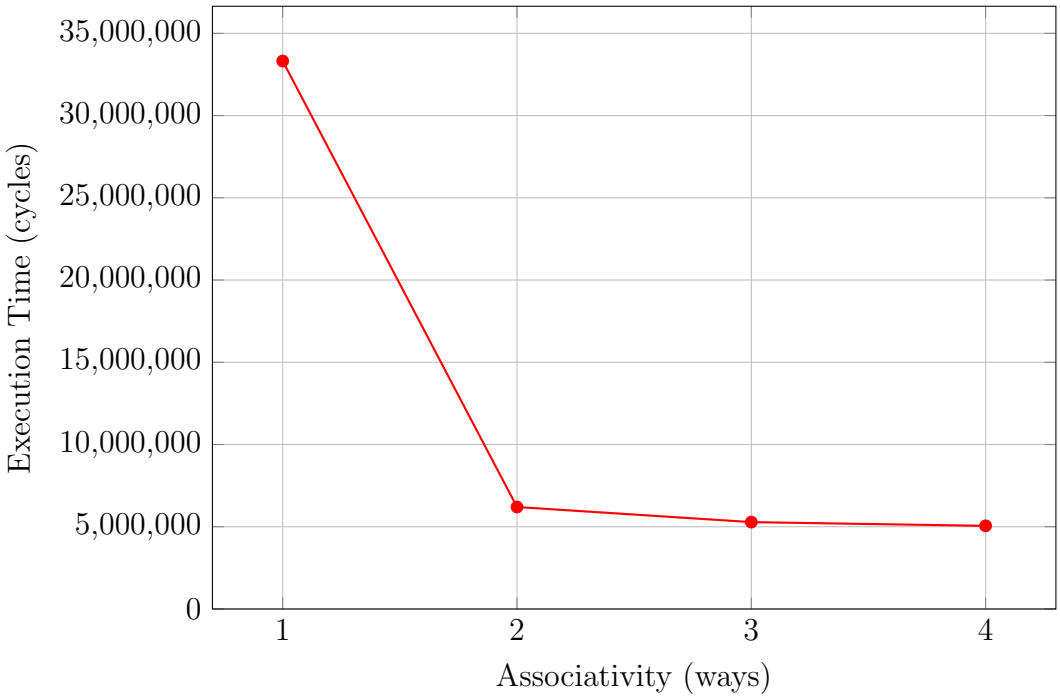


Figure 3: Cache Associativity vs. Execution Time

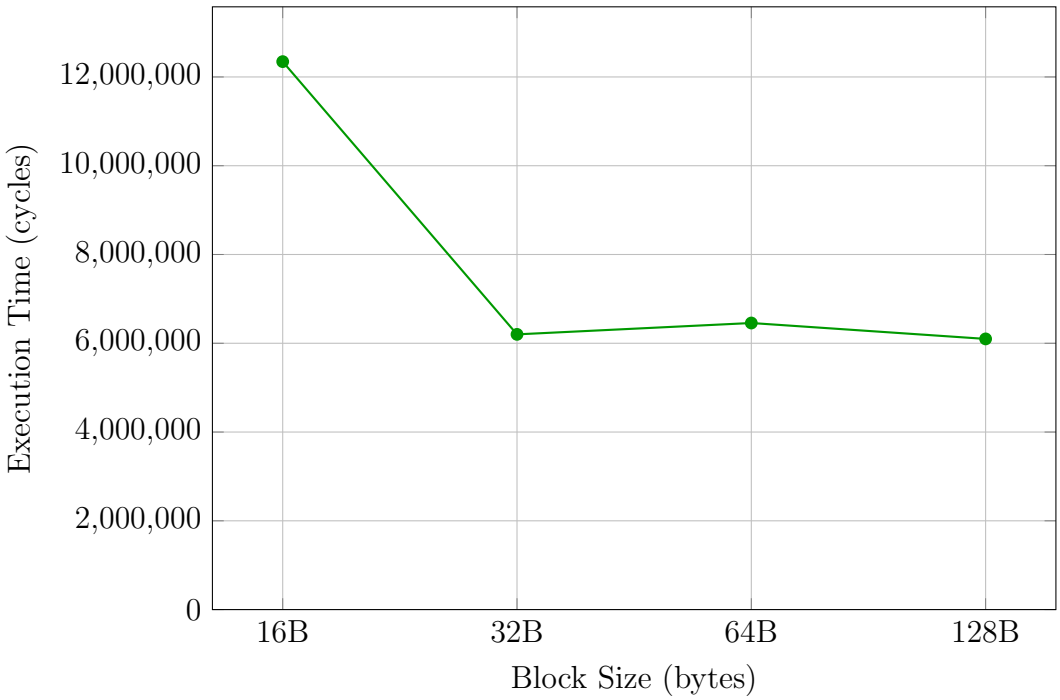


Figure 4: Block Size vs. Execution Time

- Cache size has a diminishing return on performance. While increasing cache size from 4KB to 8KB significantly improves performance, further increases beyond 8KB provide only minimal gains, indicating that the working set of this application fits comfortably within 8KB.
- Block size exhibits a non-linear relationship with performance. The 32B block size provides the best performance, leveraging spatial locality effectively. Larger block sizes introduce overhead and cache pollution, leading to decreased performance.
- Core performance varies despite similar instruction counts. Core 2 experiences higher miss rates and execution times, while Core 3 suffers from increased idle cycles due to bus arbitration prioritization, leading to higher wait times.

In conclusion, this analysis underscores the importance of tuning cache parameters to match workload characteristics. The most substantial performance improvements can be achieved by using higher associativity (preferably 8-way), a moderate block size of 32B, and a cache size of at least 8KB. Optimizing these parameters can lead to significant performance gains for the application.

6 Interesting Traces

Objective

This trace demonstrates **false sharing** where multiple cores access different words in the same cache block, causing unnecessary coherence traffic.

Trace Details

Assuming a 64-byte cache block ($b = 6$), the following accesses map to the same block:

- **Core 0:**

```
W 0x1000
W 0x1004
```

- **Core 1:**

```
R 0x1020
W 0x1024
```

- **Core 2:**

```
R 0x1040
W 0x1044
```

- **Core 3:**

```
R 0x1008
W 0x1028
```

All accesses lie within the same 64-byte cache block (0x1000 to 0x103F).

Coherence Behavior

The false sharing causes:

- Invalidations and cache-to-cache transfers between cores.
- Unnecessary bus traffic despite no actual data sharing.
- Reduced performance due to excessive coherence protocol activity.

Simulation Parameters

- $s = 0$, $E = 4$, $b = 6$ (64-byte block)
- 4 cores with MESI protocol, write-back and write-allocate cache policies

7 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.