L1 Cache Simulator

Rajat Soni 2023CS10229
Krish Bhimani 2023CS10712

April 30, 2025

Contents

1 Design and Software Architecture

1.1 Encapsulation of Functionality

1.1.1 Core Struct

```
// Data structure for a core
typedef struct core {
    int core_id; // Core Number
    ll ct_write_instructions = 0; // Count of write instructions
    ll ct_read_instructions = 0; // Count of read instructions
    ll ct_cache_hits = 0; // Count of cache hits
    ll ct_cache_misses = 0; // Count of cache misses
    ll ct_execution_cycles = 0; // Count of execution cycles
    ll ct_idle_cycles = 0; // Count of idle cycles
    ll ct_cache_evictions = 0; // Count of cache evictions
    ll ct_writebacks = 0; // Count of writebacks
    ll ct_invalidations = 0; // Count of invalidations

    ll wait_cycles = 0; // Cycles remaining for the core to finish its
        operation
    int s = 0;
    int b = 0;
    int E = 0;
} core;
```

1.1.2 Set and Cache Structs

```
// Enum for all states of a block
enum mesi_state { I, M, S, E };

// Enum for all messages on the bus
enum mess {BusRd, BusRdx}; // BusRdx is RWITM

// Data structure for a block in the cache
typedef struct block {
    ll data; // 32 bit data
    ll tag;
    ll last_used = -1; // for LRU implementation
    bool dirty_bit = 0;
    bool valid_bit = 0;
    mesi_state state = I;
} block;

// Data structure for a set in the cache (associativity)
typedef struct set {
    std::vector<block> set; // Element of vector is [tag, block(data)]
} set;

// Data structure for L1 cache
typedef struct L1cache {
    std::vector<set> table; // maps index to set
    int s = 0;
    int b = 0;
    int E = 0;
} L1cache;
```

1.1.3 MESI Data Bus Struct

```
// Data structure for the data bus
typedef struct mesi_data_bus {
    ll data;
    ll address;
    int core_id; // core id
    mesi_state next_state; // state of block in the cache after operation
    mess message; // type of the transaction (read/write)
    bool is_busy = false;
    ll wait_cycles = 0;
    std::vector<core*> cores; // vector of cores as per core id
    std::vector<L1cache*> caches;
} mesi_data_bus;
```

1.2 Assumptions

1. During each cycle, all cores attempt to execute their respective instructions, with priority increasing as core id decreases.
2. For a write or read hit, the MESI states are updated within the same cycle, and these updates may be visible to other cores during the same cycle.
3. In the event of a miss, i.e., when the data bus is utilized, the states of snooping blocks are updated at the start of the transaction, while the state of the main block (newly inserted) is updated after the MESI bus becomes available.
4. The execution cycles of snooping cores are included when they perform writebacks.
5. Idle cycles for a core are defined as the cycles during which the core is waiting for the bus to become available and cannot proceed with executing its next instruction. All other cycles are considered execution cycles.
6. The number of bus invalidations for a core is defined as the count of instances where blocks are set to the `I` state due to an invalidate message on the bus.
7. When a dirty block is evicted, both the current core and the MESI bus remain busy for an additional 100 cycles. These cycles are included in the execution cycles of the current core.
8. The parameter `b` (block bits) represents the number of bits used to index a block, including to the 2 offset bits. Therefore, a total of `b` bits are used for block indexing.
9. When a bus invalidate message is issued, the corresponding block transitions to the `I` state immediately.

1.3 Flow Chart of Simulator

Below is the flowchart of the simulator. The simulator is designed to run in a single thread. The MESI bus is also implemented as a single-threaded component, with each core waiting for its turn to access the bus. The simulator handles all cores and their interactions with the MESI bus sequentially, ensuring that each core's operations are processed in a controlled manner.

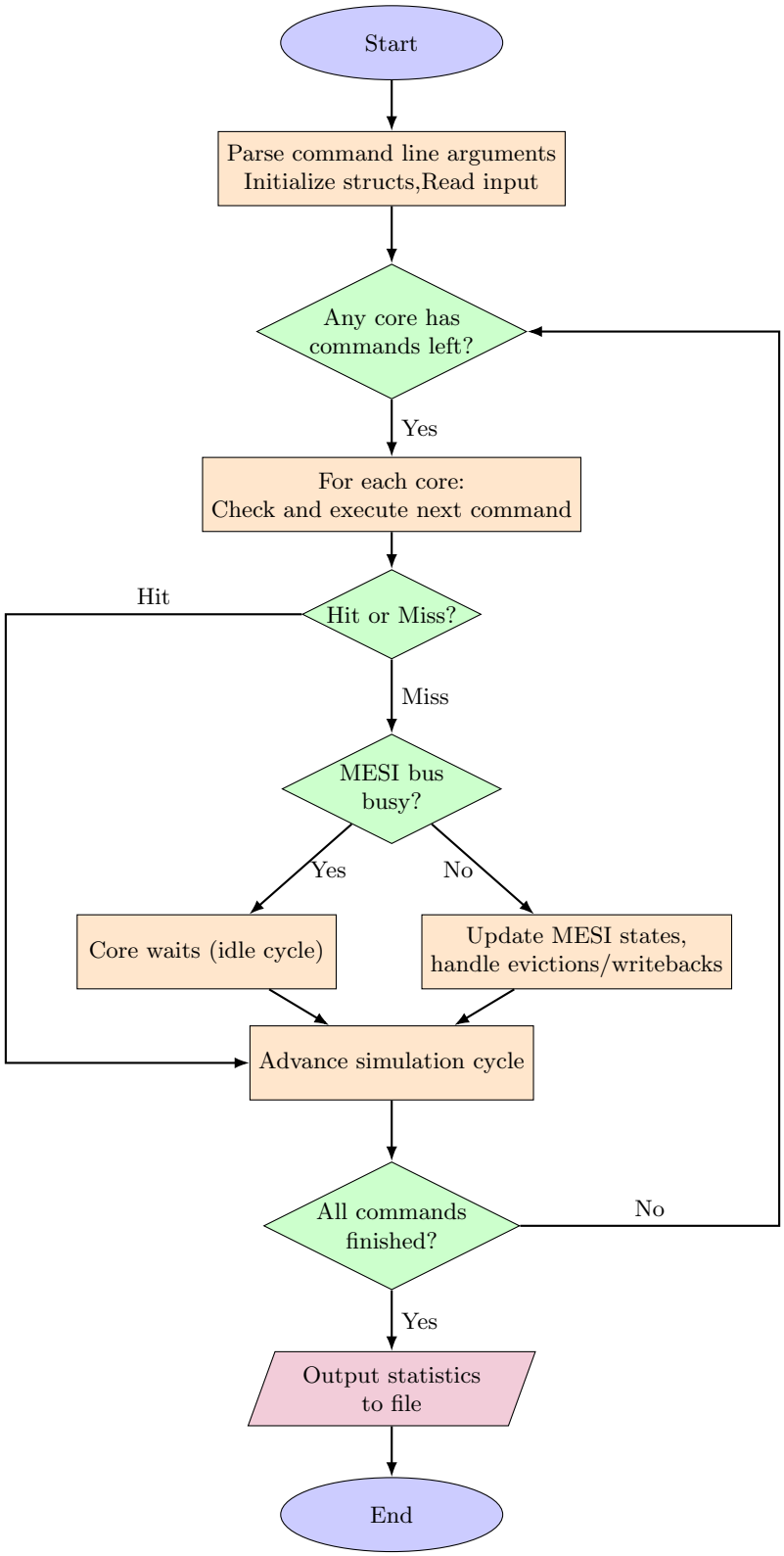

Figure 1: Flowchart of the MESI-based Multi-core Cache Simulator

2 Experimental Results

2.1 Simulation Outputs with Default Parameters

The simulator was run 10 times with default parameters: 4KB 2-way set associative L1 cache per processor with a 32-byte block size (s=6, E=2, b=5). This was done for the app1 benchmark to analyze the stability and consistency of the simulator.

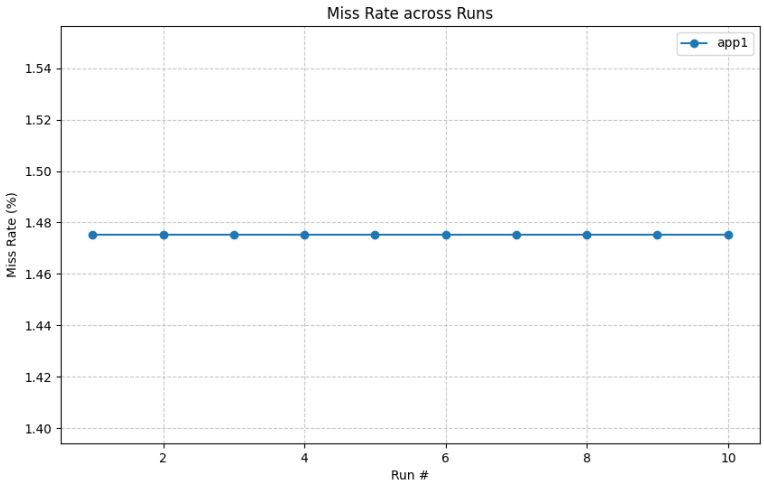

Figure 2: Cache Miss Rate Distribution Across Repeated Runs

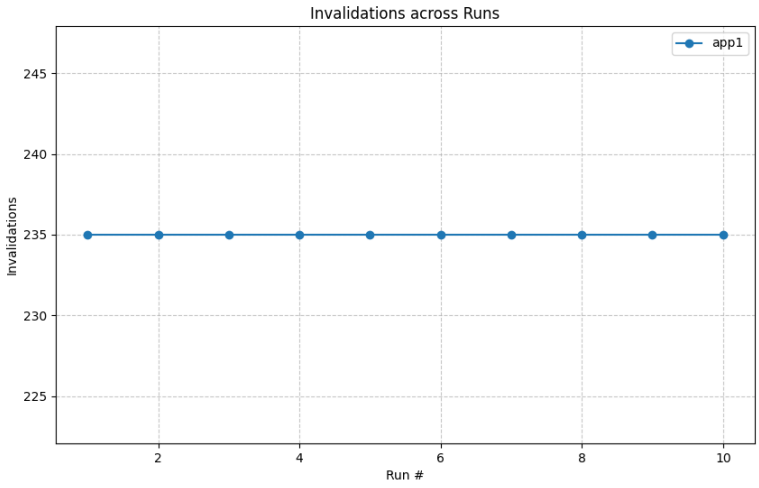

Figure 3: Bus Invalidations Distribution Across Repeated Runs

The results from multiple runs show that the simulator produces deterministic outputs. The miss rates, number of invalidations, and execution times for each run were identical. This consistency is expected because:

- The simulator uses a deterministic LRU replacement policy
- The MESI protocol state transitions follow well-defined rules
- Core prioritization is strictly based on core ID ordering

- The main simulation loop processes events in the same order for each run

This determinism is beneficial for reproducibility and validation of the simulator's functionality. Despite the complex interactions between multiple cores and the shared bus, the implementation maintains complete predictability across different runs with the same parameters.

2.2 Maximum Execution Time Analysis

The maximum execution time is calculated as the total number of cycles required for all cores to complete their assigned instructions. We conducted experiments varying three key cache parameters (cache size, associativity, and block size) while keeping the others at their default values.

2.2.1 Effect of Cache Size

We varied the cache size from 2KB to 16KB (by changing the number of set bits s from 5 to 8) while keeping associativity at 2-way and block size at 32 bytes.

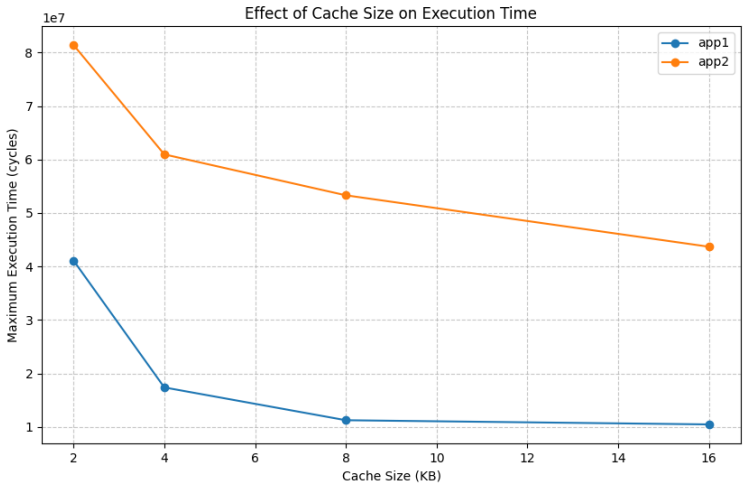

Figure 4: Effect of Cache Size on Maximum Execution Time

Observations:

- Increasing cache size generally reduces execution time due to lower miss rates
- The improvement follows a logarithmic pattern with diminishing returns
- App1 shows more significant improvements from 2KB to 4KB than from 4KB to 8KB
- App2 exhibits a more gradual improvement across all cache sizes

2.2.2 Effect of Associativity

We varied the associativity from direct-mapped (1-way) to 8-way while keeping cache size at 4KB and block size at 32 bytes.

Observations:

- Direct-mapped caches (1-way) perform significantly worse due to conflict misses
- The largest performance improvement occurs when moving from 1-way to 2-way associativity
- Beyond 4-way associativity, returns diminish substantially for both applications
- This suggests that most conflict misses are resolved with modest associativity

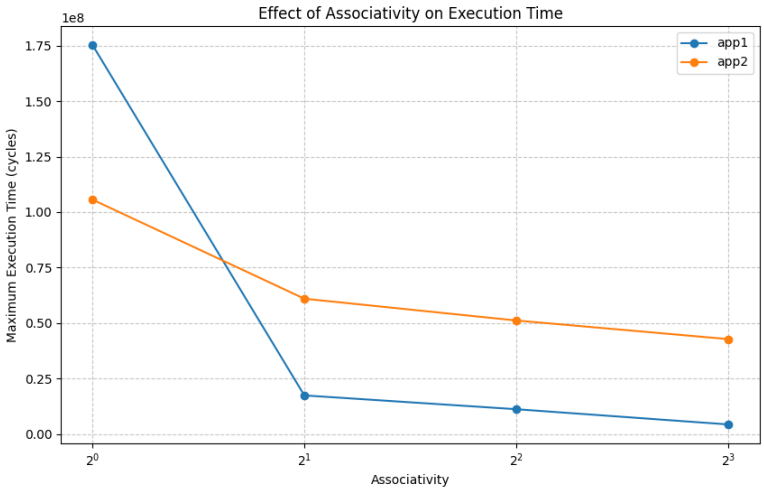

Figure 5: Effect of Associativity on Maximum Execution Time

2.2.3 Effect of Block Size

We varied the block size from 16B to 128B (by changing block bits b from 4 to 7) while keeping cache size at 4KB and associativity at 2-way.

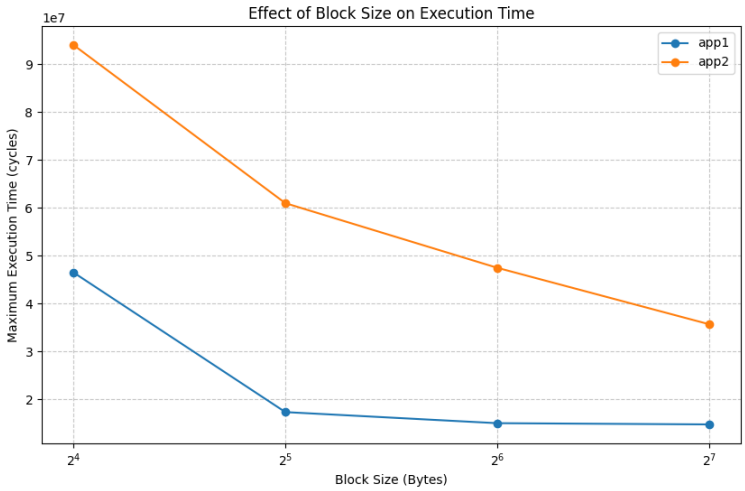

Figure 6: Effect of Block Size on Maximum Execution Time

Observations:

- Larger block sizes initially improve performance by exploiting spatial locality
- However, beyond 64B, performance may degrade due to increased bus traffic from fetching larger blocks
- App1 shows optimal performance at 32B block size, suggesting its working set benefits from medium-sized blocks
- App2 continues to benefit from larger blocks up to 64B, indicating better spatial locality

- The MESI protocol overhead increases with larger blocks due to more invalidations when sharing larger memory regions

Overall, these experiments demonstrate the trade-offs involved in cache parameter selection. The optimal configuration depends on the specific memory access patterns of the application. For general performance, a medium cache size (4-8KB) with 2-way or 4-way associativity and 32B-64B block size provides good results across tested applications while balancing the hardware complexity.

3 Test Cases

3.1 Test Case 1: False Sharing

Core 0	Core 1	Core 2	Core 3
W 0x01008000	W 0x01008000	W 0x01008000	W 0x01008000
W 0x02008000	W 0x02008000	W 0x02008000	W 0x02008000
W 0x01008000	W 0x01008000	W 0x01008000	W 0x01008000
W 0x02008000	W 0x02008000	W 0x02008000	W 0x02008000
R 0x03008000	R 0x03008000	R 0x03008000	R 0x03008000

Table 1: Input for 4 Cores

The simulator exhibits a 100% miss rate due to false sharing. When any core writes to a value, it invalidates all other shared values in the same cache block. This forces other cores to fetch data from memory instead of utilizing shared cache data, significantly impacting performance.

3.2 Test Case 2: Coherence test

Core 0	Core 1	Core 2	Core 3
R 0x10000	R 0x10000	R 0x10000	W 0x10000
R 0x10004	W 0x10000	-	-

Table 2: Input for 4 Cores

Core 0 initiates a read operation at memory address 0x10000, resulting in a cache miss. Subsequently, Core 1 performs a read at the same address, benefiting from cache coherence due to data sharing, resulting in a cache hit for Core 0 upon its subsequent read, driven by spatial locality. However, Core 1 writes to the address, invalidating Core 0's cached data. When Core 2 attempts to read, Core 1 must write the modified data back to memory to allow Core 2's read. Finally, Core 3 reads from the address and modifies its content, thereby invalidating Core 2's cached data. This sequential process illustrates the dynamics of cache coherence management in a multi-core environment.

3.3 Test Case 2: LRU test

Core 0	Core 1	Core 2	Core 3
R 0x30000	-	-	-
R 0x31000	-	-	-
R 0x32000	-	-	-
R 0x30000	-	-	-
R 0x32000	-	-	-

Table 3: Input for 4 Cores

This test case is designed to evaluate the Least Recently Used (LRU) replacement policy in a 2-way set-associative cache. Core 0 begins by reading from three distinct addresses: 0x30000, 0x31000, and 0x32000. Assuming these addresses map to the same cache set, the first two reads (0x30000 and 0x31000) fill the two available ways in the set. When the third read (0x32000) occurs, the cache set is full, so the LRU policy is triggered. The block corresponding to 0x30000, being the least recently accessed, is evicted to make room for 0x32000.

Subsequently, Core 0 reads from 0x30000 again. Since it was previously evicted, this results in a cache miss, and another eviction occurs—this time replacing the least recently used block among 0x31000 and 0x32000. Finally, the read to 0x32000 checks whether it was retained in the cache or evicted in favor of 0x30000 during the previous step. This sequence verifies that the LRU mechanism correctly tracks usage history and evicts the least recently used block when the associativity limit is exceeded.