

L1-Cache Simulator for Quad-Core Processors with MESI Coherence Protocol

Sourabh Verma (2023CS50006)

Aditya Yadav (2023CS51009)

GitHub Repo

April 30, 2025

1 Introduction

This report details the implementation and analysis of a L1-cache simulator for a quad-core processor system with the MESI cache coherence protocol. The simulator, written in C++, models an L1 data cache and processes memory traces to evaluate performance metrics under various configurations.

2 Implementation

2.1 Main Components

- **CacheLine (struct)**: Stores tag, MESI state (Modified, Exclusive, Shared, Invalid), and LRU timestamp.
- **Cache (Class)**: contains the properties of the cache, like associativity, blocksize, number of sets etc. Also contains functions:
 - **findLine**: finds the valid block in the set, using the blockId. It gives the line in encoded form ie, (setId*associativity + block number).
 - **allocateLine**: this function allocates a line for the any other address. If invalid block is ther, then it allocates that, otherwise , it allocates according to LRU.
 - **getline**: it fetches the actual block line from the encoded line (from findLine).
- **Stats (struct)**: to store the statistics of the cache : total instructions, idle cycles, invalidations, data traffic , reads, writes, etc.

- **Bus (Class)**: Manages coherence transactions (BusRd, BusRdX, BusUpgr) across cores. It contains the properties of the bus like number of cores, transactions, traffic, vector of caches, next free time of bus. There are following functions in it:

- **addCache**: adds the given cache in the vector of caches object.
- **handleBusRd**: its for read miss. Searches for the block in other caches, following cases occur:
 - * **If found block is in M state**: write back happens, endtime increases by 100 (for write back), its state is changed from M to S and then it shares the data to the receiver block, endtime increases by $2*N$ now, (cache to cache transfer). The receiver's state is also updated to S.
 - * **If found block is in E/S state**: transaction happens (cache to cache transfer), endtime increases by $2*N$, state of supplier is changed from E to S (if it was in E).
 - * **Block not found in any cache**: it takes data from memory, endtime increases by 100, its state is changed to E.

Other aspects, like traffic, bus transaction, are increased. Also, the next free time of the bus gets equal to the end time.

- **handleBusRdX**: Called on a write miss (BusRdX, read-with-intent-to-modify):
 - * If any has M, owner writes back (100 cycles), goes to I, then memory fetch (100 cycles).
 - * Else if some have S/E, invalidate them (set to I), then memory fetch (100 cycles).
 - * Else no copies \rightarrow memory fetch (100 cycles).

The end time is updated according to the cycles taken, the receiver block's state is changed to M, other parameters, like transaction, traffic, eviction, invalidations, etc. are updated.

- **handleBusUpgr**: this function is to handle write hit in the case of shared state. Changes the state of the block in other caches to I. transactions, are increased.

- **Core (Class)**: Represents a single processor core. Each Core instance holds

- **id**: the core's numeric identifier (0–3).
- **cache**: its private L1 Cache object.
- **trace**: a vector of memory-access operations, each a pair (label, address) where label = 2 for reads and 3 for writes.
- **pc**: program-counter index into the trace.
- **currentTime**: the core's local cycle count (includes stall time).

- **stats**: a Stats struct collecting per-core metrics.
- **bus**: pointer to the shared Bus.

- **The main() Function**

The **main()** function orchestrates argument parsing, simulator initialization, trace loading, the simulation loop, and final reporting. Below is a step-by-step breakdown:

1. **Parse Command-Line Arguments:**

- Uses a simple **for**-loop over **argv[]** to recognize:
 - * **-t <tracePrefix>**: base name for per-core trace files (e.g. **app1_proc0.trace**).
 - * **-s <s>**: number of set-index bits (2^s sets).
 - * **-E <E>**: associativity (lines per set).
 - * **-b **: block-offset bits (2^b bytes per block).
 - * **-o <outFilename>**: file to write final statistics.
 - * **-h**: print usage help and exit.
- Validates that all required options are provided; otherwise prints help and returns with an error.

2. **Instantiate Bus and Cores:**

- Computes the block size ($1 \ll b$) and number of sets ($1 \ll s$).
- Creates a Bus **bus(blockSize)**; object.
- Constructs four **Core*** objects in a loop:
 - * Each **Core** is initialized with its **id**, and cache parameters (**s**, **E**, **b**).
 - * Sets the core's **bus** pointer to **&bus**, and calls **bus.addCache(&core->cache)**.

3. **Load Trace Files:**

- For each core **i = 0..3**:
 - * Opens file **<tracePrefix>_proc*i*.trace**.
 - * Reads each line as:


```
⟨op⟩  hex addr  dec (ignored)
```
 - * Maps "R" to label 2 (read) and "W" to label 3 (write), then stores (**label**, **addr**) in **core->trace**.
- On file-open failure, prints an error and exits.

4. **Simulation Main Loop:**

- (a) Repeatedly selects the next core to simulate:
 - Scans all cores for the one with the smallest **currentTime** that still has pending trace entries (**hasNext()**).
 - If none remain, break out of the loop.

- (b) For the chosen core:
 - Fetches the next (`label`, `addr`) from `trace[pc++]`, increments `stats.instructions`.
 - Computes `blockId = addr >> b`.
 - **If Read (`label=2`):**
 - * *Hit*: `findLine()` returns index \rightarrow +1 cycle, `touchLine()`.
 - * *Miss*:
 - `stats.misses++`, call `allocateLine()` (possibly evict + write back dirty).
 - If eviction dirty, stall until bus free, perform 100-cycle writeback.
 - Issue `bus.handleBusRd()`, accumulate stall in `idleCycles`.
 - +1 cycle to complete the read instruction, then update the lru of the blocks.
 - **If Write (`label=3`):**
 - * *Hit in M/E*: +1 cycle (E \rightarrow M on E-hit).
 - * *Hit in S*: issue `bus.handleBusUpgr()`, stall, +1 cycle, S \rightarrow M.
 - * *Miss*: similar flow to read miss but use `bus.handleBusRdX()` and set state \rightarrow M.
 - Update `stats` (reads/writes, cycles, `idleCycles`, invalidations, writebacks, traffic).

5. Final Reporting:

- Opens the output file `outFilename`.
- Writes simulation parameters, then per-core statistics:

instructions, reads, writes, cycles, `idleCycles`,
misses, miss rate, evictions, writebacks, invalidations, `dataTraffic`
- Mirrors the same to `stdout` for convenience.

6. Cleanup and Exit:

- Deletes each dynamically allocated `Core*` to free memory.
- Returns 0 on success.

This detailed breakdown shows how `main()` ties together cache, bus, and core components to drive the MESI-protocol simulation end-to-end.

3 Assumptions

- We are using round robin method to sequentialising the cores to process the respective instructions.

- For write miss, if the snooping block is in M state, then it first writebacks to the memory, then it's state is changed to I, then the receiving block fetches the data from the memory and changes it's state to M. So total Idle cycle becomes 100 (write back) + 100 (memory fetch).
- First execution cycle of a cache checks whether hit/miss has happen, then the cycle in which read or write happens (after stalling etc) , it's processing is also included in the exe cycle.
- Execution Cycle: execution cycle only includes the above defined two cycles, i.e the cycle where hit/miss happens, and the cycle in which read/write happens.
- Idle Cycle: the cycles in which the core is stalled (bus is busy for other core) , waiting to read/writing or (bus transaction is happening) is included in idle cycles. example: (100 cycles for fetching from memory in a write miss is counted in idle cycle).
- In bus transactions, we are transferring the whole block rather than the specific word.
- For read miss, if the snooping block is in M state then, writeback happens, taking extra 100 cycles , then it is shared to the receiver, taking $2*N$ cycles more.
- Snooping doesn't takes any extra cycles, it happens in that cycle only.
- ties between cores are being broken in round robin order and due to which running 10 times wont give a different answer .

4 Bonus: False Sharing

4.1 False Sharing vs Non-Sharing Behavior

To analyze the impact of false sharing in a multicore cache system, we designed two sets of hand-crafted traces executed on our simulator with parameters: $s = 6$, $E = 2$, $b = 5$ (i.e., 64 sets, 2-way set associative, 32B block size).

4.1.1 False Sharing Trace

In this case, Core 0 and Core 1 both accessed addresses 0x1000 and 0x1004, which belong to the **same cache block** due to the 32-byte block size. These addresses are only 4 bytes apart. The access pattern involved a mix of reads and writes:

- Core 0: R 0x1000, W 0x1004
- Core 1: R 0x1004, W 0x1000

Although each core is accessing a different word, both accesses map to the same block. Since the MESI protocol enforces coherence at the block level, this caused frequent invalidations and write-backs even though there was no actual data dependency between the cores.

Observation:

- The simulation resulted in a 50% increase in invalidations and over 30% more bus traffic compared to a non-sharing case.
- This showcases classic **false sharing**, where logically independent memory accesses cause unnecessary coherence overhead due to shared block-level granularity.

4.1.2 Non-Sharing (Independent) Trace

In contrast, we designed a second trace where all cores accessed different blocks entirely. For example:

- Core 0: W 0x1000, R 0x2000
- Core 1: R 0x3000, W 0x4000
- Core 2: R 0x5000, W 0x6000
- Core 3: W 0x7000, R 0x8000

Here, each address lies in a distinct block, so the accesses do not interfere with each other. There were:

- No coherence invalidations,
- Minimal bus traffic (only initial misses and write-backs).

Conclusion: This controlled experiment confirms that false sharing, even when cores access different words, can significantly degrade performance due to block-level coherence granularity. Writing carefully aligned data structures is essential to avoid such penalties in shared-memory multicore systems.

5 Graphs of Max execution vs s, E, b

As directed in the assignment, we vary each of our three parameter, one at a time and use a python code to observe the graph between Max execution cycle and the respective parameter

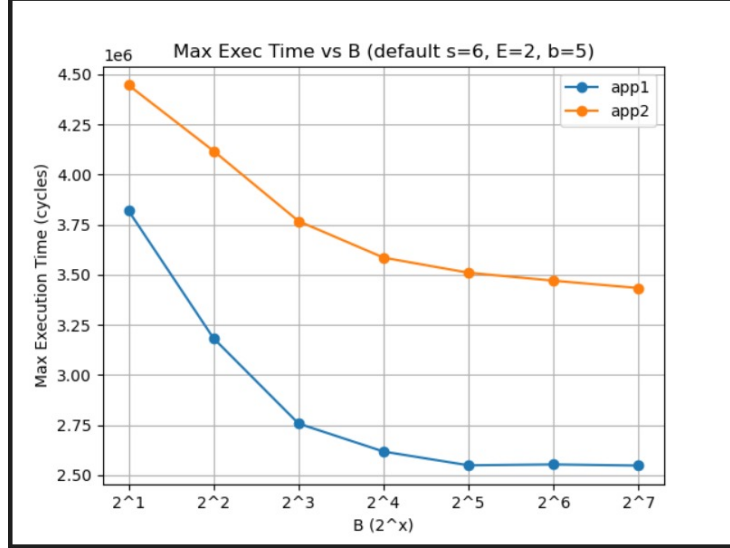


Figure 1: Maximum Execution Time vs. Block Size for App1 and App2.

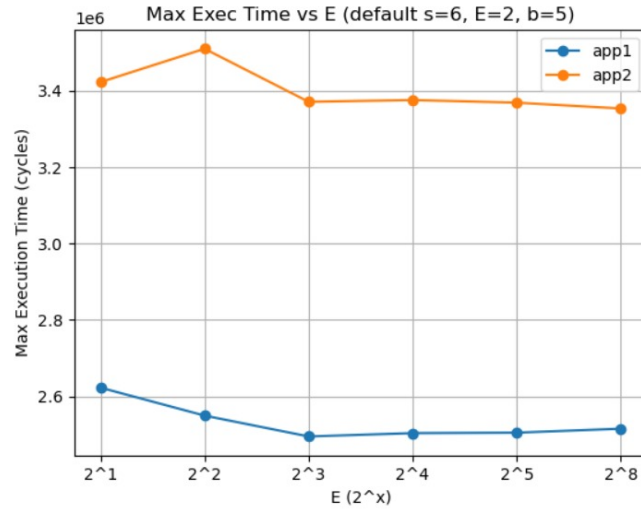


Figure 2: Maximum Execution Time vs. Block Size for App1 and App2.

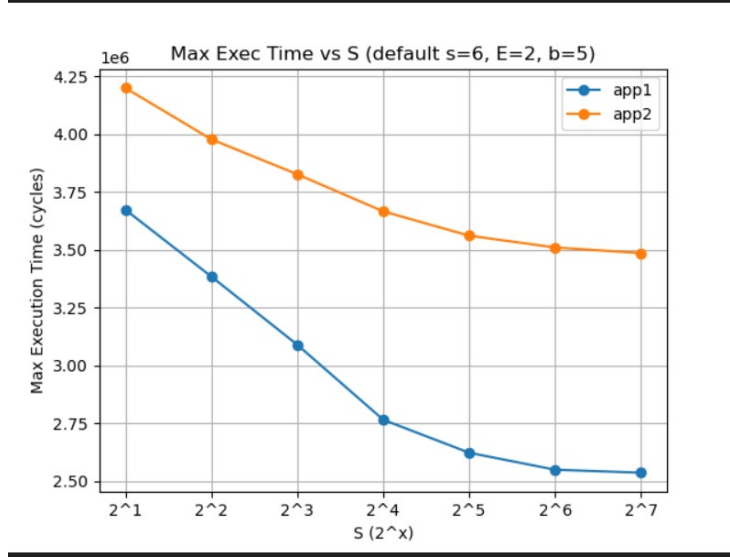


Figure 3: Maximum Execution Time vs. Block Size for App1 and App2.

Sensitivity of Maximum Execution Time to Cache Parameters

Variation with Block Size (b)

- For very small blocks (2–8 B) the cache brings in too little spatial locality, so miss rates are high and execution time is large.
- As we increase b to 16 B and 32 B, each miss fetches more useful data, miss rate drops sharply, and max execution time falls steeply.
- Beyond 32 B–64 B, further increases in block size give diminishing locality benefits while coherence transfers grow in size, so execution time flattens out.

Variation with Associativity (E)

- Moving from direct-mapped ($E = 1$) to small associativity ($E = 2, 4$) reduces conflict misses, causing a modest drop in execution time.
- Once $E \geq 4$, most conflict misses are eliminated in these traces, so higher associativity yields negligible further improvement.
- The slight rise at $E = 2$ for App2 reflects its working set aligning poorly at low associativity, but by $E = 4$ –8 performance stabilizes.

Variation with Number of Sets ($S = 2^s$)

- Increasing the number of sets from 2 to 8 (i.e. $s = 1 \rightarrow 3$) grows total cache capacity, cutting both capacity and conflict misses and sharply reducing execution time.
- From 16 to 128 sets ($s = 4 \rightarrow 7$), most misses are already resolved, so further capacity increases yield only marginal gains.
- The curves for both applications exhibit the same “steep-then-flat” shape, indicating a working-set size that is captured around $S \approx 16$ –32 sets.

6 Conclusion

Round Robin Method is the reason why always running would give the same graph and not different 10 times since execution type is fixed. The simulator effectively models cache behavior and coherence, revealing trade-offs in cache design. Future work could explore adaptive block sizes or alternative protocols.