COL216 Assignment 3

L1 Cache Simulator for Quad-core Processors

(MESI Protocol)

Name	Entry Number
Himanshi Bhandari	2023CS10888
Mohil Shukla	2023CS10186

April 30, 2025

Github Link

Contents

1 Cache Simulator Implementation

1.1 Overview

The simulator models a multi-core system with private L1 caches and a shared bus, implementing the MESI coherence protocol. It processes memory access traces and generates performance statistics.

1.2 Class Structure

1.2.1 L1Cache Class (l1cache.cpp)

The core cache implementation with:

- **Data Members**:
  Private:
  - `vector<CacheSet> sets`: Cache storage array,
    where:
    - `struct CacheSet(defined in l1cache.h)`: Stores vector¡CacheLine¿ lines
    - `enum MESIState(defined in l1cache.h)`: Stores MESI States- INVALID, SHARED, EXCLUSIVE, MODIFIED
    - `struct CacheLine(defined in l1cache.h)`: Stores tag, MESIState ,bool dirty,lru_counter, bool empty
  - `s, E, b`: Cache parameters (set bits, associativity, block bits)
  - Blocking state variables : bool blocked: tells if the Cache is blocked (processing the work for example doing cache to cache transfer), block_cycles_left: tells ow many cycles left for the work to complete for which it is blocked, blocked_addr : address of CacheLine due to which blocking happened
  - Pending State: which state it will be in after blocking is finished

  Public: reads , writes, misses, evictions, writebacks , invalidations , idle_cycles ,execution_cycles , bus_traffic

- **Key Methods**:
  - `try_access()`: Handles read/write requests
  - `snoop()`: Processes bus transactions
  - `find_line()`: Locates cache lines
  - `find_lru()`: LRU replacement policy

1.2.2 Bus Class(bus.cpp)

Models the shared bus with:

- **Data Members**:
  - `current`: current Bus transaction
    where:
    - `BusTransactionType (defined in bus.h)`: Stores NONE, Transaction type : BusRd, BusRdX, BusUpgr
    - `struct BusTransaction (defined in bus.h)`: Stores type: type of Bus stransaction, addr : address which has started the transaction, source_core : which core started the transaction,transfer_cycles_left: cycles left to complete the transaction

- **Key Methods**:

- `start()`: Begins new transaction
- `tick()`: Decrements timer
- `busy()`: Checks bus status if it is busy or not
- `transfer_cycles_left()`: Remaining busy cycles
- `change`: changes the state of bus from busy to free

1.2.3 Simulator Class(simulator.cpp)

Orchestrates the simulation:

- **Data Members**:
  - `vector<L1Cache> caches`: Per-core caches
  - `Bus bus`: Shared bus (instance of Class Bus)
  - `global_cycle`: Simulation time(cycle)
  - `vector<bool> done`: it is of size 4 one for each core , tells if the core is done or not i.e. are all instructions completed for that core.
  - `vector<bool> waiting_for_bus`: it is of size 4 one for each core , tells if the core is waiting for the bus.
  - `vector<std::ifstream*> tracefiles`: stores tracefiles for each core
  - `vector<std::string> next_line`: stores next line for each core
- **Key Methods**:
  - `run() : Public`: Main simulation loop
  - `print_stats(): Public`: Outputs metrics
  - `all_done(): Private`: Tells if all the 4 cores are done or not

1.2.4 main.cpp

The main function parses command-line arguments to configure cache parameters and input/output files, then initializes the Simulator with these settings. It runs the simulation and writes the results to the specified output file.

2 Cache Simulator Workflow

The simulator executes in the following sequence:

1. **Initialization** (main.cpp)
   - Parses command line arguments
   - Creates Simulator instance

     ```
     Simulator sim(s, E, b, tracebase);
     ```

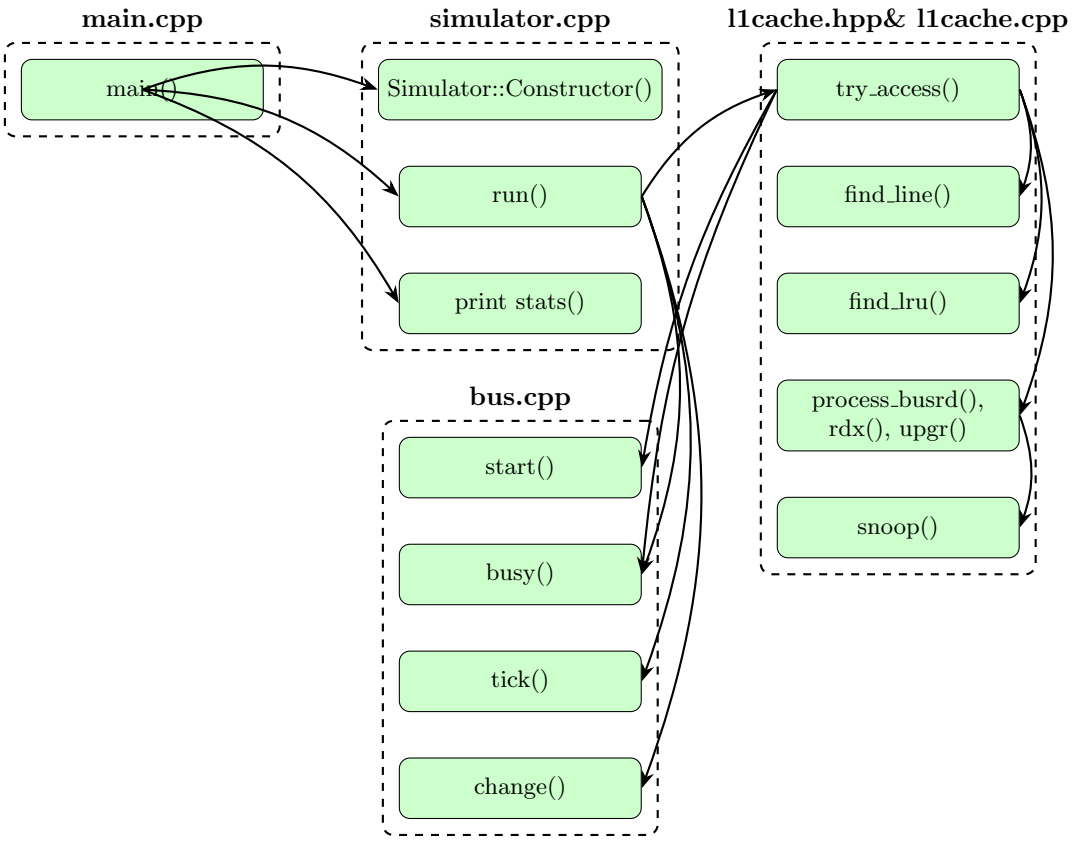
     (hence we go to **Setup Phase in simulator.cpp**)
   - once instances of L1 caches for each core is made in setup phase , sim.run() is called hence we reach to **Simulation Loop**
   - after this is done we have completed the simulation and we reach **Termination** stage
2. **Setup Phase** (simulator.cpp)
   - Initializes 4 L1 caches with the parameters given s, E,b

- Opens trace files for each core

3. **Simulation Loop** (simulator.cpp::run())
   (a) **Initialization**
      – Get line to processed for each core
      – Process(stay in while loop) till all the four cores are done
   (b) **Core Loop**
      – Processes each core sequentially from least to greatest core_id(simulator.cpp)
      – Checks completion status first
      – Handle if cache is blocked or not
   (c) **Blocked Core Handling**
      – Counts down block_cycles_left (cache miss penalty)
      – When block_cycles_left reaches 0:
         ∗ Update the state of line to the Pending state
         ∗ Update the lru counter of the Cacheline
         ∗ Change the pending state to invalid i.e. it is no more blocked and set the core that it is no more blocked
         ∗ since now bus is no more busy, set waiting_for_bus to false for each core.
      – Go to the next iteration i.e. next core
   (d) **Mark Done**
      – If next line is empty mark done
   (e) **Instruction Processing**
      – Parses trace file lines (format: "R/W 0xADDR")
      – Calls **try_access()** of lache.cpp for cache operations **(go to Core Processing)**
   (f) **Access Resolution(try_access true or false)**
      – Successful access fetches next instruction
      – Failed access (bus busy) marks core as waiting

4. **Core Processing** (l1cache.cpp::try_access())
   - Checks for cache hits
   - On miss: finds replacement line (evicts if needed)
   - Updates statistics

5. **Termination**
   - Ends when all traces complete
   - Outputs statistics (simulator.cpp::print_stats())



3 Core Algorithms

3.1 Function: `try_access()` in `L1Cache` Class

3.1.1 Introduction

The `try_access()` function is the core of L1 cache behavior in the simulator. It takes in a memory access (either read or write), determines whether it results in a cache hit or miss, and updates the cache and coherence state accordingly based on the MESI protocol. **3.1.2 Initial Steps**

- First, the **tag** and **set index** are calculated from the memory address.
- Using these, we attempt to locate a line in this core's cache set that matches the tag by calling `find_line()`.

3.1.3 Case 1: Cache Hit

If a matching line is found and is not in the INVALID state, it is considered a cache hit. In this case:

- The **LRU counter** is updated to reflect recent access.

Sub-case: Read Hit

- The line is in a valid MESI state.
- No state transition is needed; we simply increment the `read` counter.
- The function returns `true`, indicating successful access.

Sub-case: Write Hit

- If the line is in `SHARED` state:
  - First, we check if the bus is busy. If yes, we return `false` to indicate the access must wait.
  - If the bus is free, we call `process_busupgr()` to send a `BusUpgr` transaction, prompting invalidation of shared copies in other caches.
  - We increment the `invalidations` counter.
  - The line's state is updated to `MODIFIED`.
- If the line is already in `EXCLUSIVE` or `MODIFIED`, we directly write and return `true`.

Execution cycles are incremented accordingly before returning `true`.

3.1.4 Case 2: Cache Miss

If no matching valid line is found in the set, it is a cache miss.

Initial Step

- If the bus is currently busy, we return `false` to wait.
- We check for an available empty line in the cache set.
- If the set is full, we select a line to evict using LRU policy.

Eviction Handling

- If the evicted line is in the `MODIFIED` state, a writeback to memory is required.
- If evicting a `SHARED` line and exactly one other copy exists in another cache, we downgrade that copy to `EXCLUSIVE`.
- In all eviction cases, the LRU counter is updated for the selected line.

Sub-case: Read Miss

- We check all other caches for the requested block:
  - If another cache holds the block in the `MODIFIED` state, a writeback to memory is required. This involves:
    - The modified cache placing the value on the bus for other caches (latency: $2N$ cycles), and
    - Writing the value to memory (latency: 100 cycles).

    Hence a total of $2N + 100$ cycles.
    The other cache transitions to `SHARED` via snooping, and the requesting cache sets its pending state to `SHARED`.
  - If another cache has the block in the `SHARED` or `EXCLUSIVE` state, it is fetched via a cache-to-cache transfer (latency: $2N$ cycles). The other cache transitions to `SHARED` via snooping, and the requesting cache sets its pending state to `SHARED`.
  - If no other cache holds the block, it is fetched from main memory (latency: 100 cycles), and the pending state of the requesting cache is set to `EXCLUSIVE`.
- After the completion of the required cycles, the new line is inserted with the following MESI state:
  - `SHARED`, if multiple caches now hold the block.
  - `EXCLUSIVE`, if this cache is the only one holding the block.
- The read counter and execution cycles are updated, and the function returns `true`.

Sub-case: Write Miss

- We again check all other caches for the requested block:

  - If another cache holds the block in the `MODIFIED` state, a writeback to memory is required (latency: 100 cycles), followed by invalidation. The data is then read from memory (another 100 cycles), totaling 200 cycles. The pending state is set to `MODIFIED`.
  - If the block is in the `SHARED` or `EXCLUSIVE` state in another cache, all other copies are invalidated. This takes 100 cycles (to ensure memory consistency), and the pending state is set to `MODIFIED`.
  - If no other cache holds the block, it is fetched from memory (latency: 100 cycles), and the pending state is set to `MODIFIED`.
- After the required latency completes, the new line is inserted into this cache in the `MODIFIED` state.
- The write counter and execution cycles are updated, and the function returns `true`.

3.1.5 Summary of States and Transitions

This function ensures coherence using the MESI protocol:

- `MODIFIED` lines are unique and dirty.
- `EXCLUSIVE` lines are unique and clean.
- `SHARED` lines may exist in multiple caches and are clean.
- `INVALID` lines are unused or outdated.

State transitions and data movement happen through bus transactions (`BusRd`, `BusRdX`, `BusUpgr`) and snooping mechanisms handled via `process_busrd()`, `process_busrdx()`, and `process_busupgr` functions.In these functions snoop function is called for all the other cores except the one who is currently sending the information.

Overall, the `try_access()` function models detailed cache behavior and coherence logic in a multi-core system while capturing realistic latency and eviction mechanisms.

3.2 MESI State Transitions

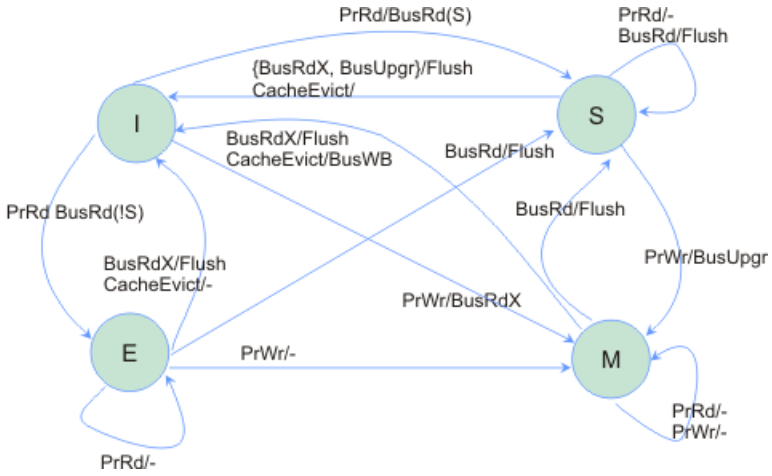


Figure 1: MESI protocol diagram: sample state transitions. See: Source

3.3 Bus Transactions and Snooping Mechanism

In a multiprocessor system using the MESI protocol, cache coherence is maintained using a shared bus through which caches communicate. This is achieved via *snooping*, where each cache monitors (or snoops) the bus to track memory accesses by other processors. The following are the key types of bus transactions involved:

- **Snoop Mechanism:**
  All caches continuously observe the bus for any transactions related to memory blocks they might hold. Based on the type of transaction and the current state of the block, each cache takes appropriate action—such as supplying data, invalidating its copy, or updating its state.
- **BusRd (Bus Read):**
  This transaction is initiated when a processor performs a read operation for a block that is not present in its cache. Other caches snoop this request:
  - If any cache holds the block in the `MODIFIED` state, it must write back the data (to memory).
  - If any cache holds the block in `SHARED` or `EXCLUSIVE`, it provides the data and transitions to `SHARED`.

  The requesting processor receives the data and stores it in `SHARED` or `EXCLUSIVE` state, depending on whether other caches had the block.
- **BusRdX (Bus Read Exclusive):**
  This transaction is issued when a processor wants to write to a block that is not in its cache. It functions similarly to `BusRd`, but with the additional goal of ensuring exclusivity:
  - All other copies of the block in other caches are invalidated.
  - The requesting cache obtains the block and transitions to the `MODIFIED` state after any necessary writeback or data fetch.
- **BusUpgr (Bus Upgrade):**
  When a processor already holds the required block in the `SHARED` state and intends to write to it, it issues a `BusUpgr` request:
  - No data transfer is needed, as the processor already has the block.
  - All other caches invalidate their copies of the block.
  - The requesting cache then transitions to the `MODIFIED` state.

3.4 Explanation of `run()` Function in `simulator.cpp`

The `run()` function simulates the execution of memory instructions by multiple processor cores, managing cache coherence, bus access, and cycle tracking. The function executes in a cycle-by-cycle manner and ensures that all cores interact correctly while accessing memory.

The simulation starts by initializing the first instruction for each core from the trace file. Each core executes memory operations (either reads or writes), and the `run()` function loops through the following steps:

- First, it checks if all cores are finished by calling `all_done()`. If not, the function continues to the next cycle.
- The function then simulates a bus cycle by calling `bus.tick()`.
- For each core, the function checks if the core has finished processing its instructions. If the core is done, it skips to the next core. If the core is blocked (e.g., waiting for data from memory), it decrements the cycle counter for that core and checks for cache access.
- If the core is waiting for the bus, the function increments its cycle counter and skips its memory operation for that cycle.
- If the core has no more instructions, it is marked as `done`, and the function continues with the next core.
- If the core is still active, the next instruction is parsed. This instruction contains an operation (read or write) and an address. The function converts the address to an unsigned integer.
- The core then accesses its cache, either hitting the cache or issuing a bus transaction if the data is not present.
- If the core was able to proceed with the operation, the next instruction is read from the trace file, and the core state is updated. If the core is still waiting for the bus, it remains blocked until the bus is available.
- After each cycle, the global cycle counter is incremented.
- The process continues until all cores are finished with their operations.

The core logic of this function ensures that memory operations are handled in parallel, and the cores synchronize their memory accesses via the bus. The function models the interaction of multiple cores sharing a memory hierarchy and the impact of cache coherence and bus contention on overall execution.

Once all cores finish their instructions, the function exits, and the simulation completes.

4 Design Decision Assumptions

- If Core 0 was blocked and on cycle 100 it was unblocked, then on cycle 100 all other cores will still the bus as busy, only on the 101st cycle will the cores find the bus to be free and the first core to execute a free bus instruction would be Core 0 itself.
- Cachelines with Invalid states are assumed as Empty, i.e. they are given the highest priority to evict and also dont count in evictions when we evict them.
- Even if the bus is busy and we arrive at a instruction which is waiting for the bus, we update the LRU COUNTER because we dont want to evict it as it will soon be used. We have implement the LRU strategy to evict from cache, i.e. the least recently used bit will be evicted if no cache line is empty or invalid.
- If Two or more cores are both going to use the bus then we give priority to the lower core.
- Bus can't be used for send signals like invalidate , when it is busy. That is, bus is considered busy while transfering data and signals , and they canonly happen one at a time.
- transfer_cycles = (2 * B/4)+100 we consider core to be busy in whole of this time similarly in write miss another modified case for 200 cycles.

- We have assumed that it takes 100 extra cycles after the miss to complete a read miss after reading from memory only after which can the next input can be processed (total 101 cycles).

5 Performance Metrics

5.1 Metrics Collected (Per Core)

The following metrics were collected for each core:

- **Reads/Writes**: The number of memory instructions executed by the core.
- **Misses**: The total number of cache misses encountered.
- **Miss Rate**: The ratio of cache misses to total memory accesses, calculated as:

$$\text{Miss Rate} = \frac{\#\text{Misses}}{\#\text{Accesses}} \times 100\%$$

- **Idle Cycles**: The cycles in which this core is waiting for the bus which was issued by some other Core.
- **Execution Cycles**: The cycles in which this core is performing tasks, i.e. either cache hits or misses are happening or the core is waiting for the bus which was issued by this own Core.
- **Evictions**: The count of (Non Invalid) cache lines evicted
- **Writebacks**: The number of lines written back to memory.
- **Invalidations**: The number of times this Core has made another Core(s) Invalid by sending instructions through the bus.
- **Bus Traffic**: The total data that was transferred through the bus which was issued by this core (Cache-Cache, Mem-Cache, Cache-mem).
  (Note that according to our assumptions we don't count the bus traffic on the senders side we only count it on the Core which calls the bus)

5.2 Bus Statistics

The following metrics were collected for the bus:

- **Bus transaction**: The number of times the Bus is used either for snooping or data transfer.
- **Traffic**: The total amount of data transferred on the bus, measured in bytes. This is basically the sum of all the individual bus traffics.

6 Testing

6.1 Check for Hits

Table 1: Check for Hits

	Core0	**Core1**	**Core2**	**Core3**
Instruction 1	R 0x00000000	W 0x9001	W 0x9000	W 0x9003
Instruction 2	R 0x00000001			
Instruction 3	W 0x00000000			
Misses	1	1	1	1
Hits	2	0	0	0

- **Core 0 - Instruction 1:** A `Read` operation to a memory block not present in the cache results in a **read miss**.
- **Core 0 - Instruction 2:** A second `Read` operation to the same memory block now results in a **read hit**, as the block was fetched in the previous instruction.
- **Core 1 - Instruction 1:** A `Read` operation to the same memory block leads to a **read miss**, since the block is not yet present in Core 1's cache.
- **Core 0 - Instruction 3:** A `Write` operation to the same memory block now results in a **write hit**, as the block is still present in Core 0's cache.
- **Core 2 - Instruction 1:** A `Read` operation to a different memory block not currently cached, resulting in a **read miss**.
- **Core 3 - Instruction 1:** A `Write` operation to a new memory block also leads to a **write miss**.

6.2 Check for Misses

Table 2: Check for Misses

	Core0	**Core1**	**Core2**	**Core3**
Instruction 1	R 0x00000000	W 0x00000001	R 0x00100001	W 0x9003
Instruction 2	R 0x00000001	R 0x00000001	R 0x00100001	
Instruction 3	W 0x00000000			
Misses	2	2	1	1
Hits	1	0	1	0

- **Core 0 - Instruction 1:** A `Read` operation to a memory block not present in the cache results in a **read miss**.
- **Core 0 - Instruction 2:** A second `Read` operation to the same memory block now results in a **read hit**, as the block was fetched in the previous instruction.
- **Core 1 - Instruction 1:** A `Write` operation to the same memory block leads to a **write miss**, since the block is not yet present in Core 1's cache. But writing to that address invalidates the memory block of Core 0.
- **Core 0 - Instruction 3:** A `Write` operation to the same memory block now results in a **write miss**, as the block is now not present in Core 0's cache as it was invalidated by Core 1. Now this write invalidates the memory block of Core 1.
- **Core 1 - Instruction 2:** A `Read` operation to the same memory block, whch is now invalidated resulting in a **read miss**.
- **Core 2 - Instruction 1:** A `Read` operation to a new memory block which is not yet cached in this core leads to a **Read miss**.
- **Core 2 - Instruction 2:** A `Read` operation to the same memory block which leads to a **Read hit**.
- **Core 3 - Instruction 1:** A `Read` operation to a new memory block which is not yet cached in this core leads to a **Read miss**.

Table 3: Check for Eviction 1

	Core0	Core1	Core2	Core3
Instruction 1	R 0x00000000	W 0x9001	W 0x9002	W 0x9003
Instruction 2	R 0x00000800			
Instruction 3	R 0x00001000			
Evictions	1	0	0	0

6.3 Check for Proper Eviction

6.3.1 Evictions Testcase 1

- **Core 0 - Instruction 1:** A `Read` operation to a memory block not present in the cache results in a **read miss**.This memory block is brought to the cache.
- **Core 0 - Instruction 2:** A `Read` operation to a memory block not present in the cache results in a **read miss**.This memory block is brought to the cache in the same set as previous.
- **Core 0 - Instruction 3:** A `Read` operation to a memory block not present in the cache results in a **read miss**.But now the set is full , hence eviction happens. This memory block is then brought to the cache.
- **Core 1 - Instruction 1:** A `Write` operation to a memory block not present in the cache results in a **write miss**.This memory block is brought to the cache.
- **Core 2 - Instruction 1:** A `Write` operation to a memory block not present in the cache results in a **write miss**.This memory block is brought to the cache.
- **Core 3 - Instruction 1:** A `Write` operation to a memory block not present in the cache results in a **write miss**.This memory block is brought to the cache.

6.3.2 Evictions Testcase 2

Table 4: Check for Eviction 2

	Core0	Core1	Core2	Core3
Instruction 1	R 0x00000000	W 0x00000000	W 0x9002	W 0x9003
Instruction 2	R 0x00000000			
Instruction 3	R 0x00001000			
Instruction 4	R 0x00000800			
Evictions	0	0	0	0

- **Core 0 - Instruction 1:** A `Read` operation to a memory block not present in the cache results in a **read miss**.This memory block is brought to the cache.
- **Core 0 - Instruction 2:** A `Read` operation to a memory block already present in the cache results in a **read hit**.
- **Core 1 - Instruction 1:** A `Write` operation to a memory block not present in the cache results in a **write miss**.This memory block is brought to the cache.But this invalidates the Core 0 hence now Core0 cache is back to empty.
- **Core 0 - Instruction 3:** A `Read` operation to a memory block not present in the cache results in a **read miss**.This memory block is brought to the cache.

- **Core 0 - Instruction 4:** A `Read` operation to a memory block not present in the cache results in a **read miss**.But now the set is not full as due to invalidation above the core became empty so now only 0x00001000 was in the cache set, hence no eviction happens. This memory block is then brought to the cache.
- **Core 2 - Instruction 1:** A `Write` operation to a memory block not present in the cache results in a **write miss**.This memory block is brought to the cache.
- **Core 3 - Instruction 1:** A `Write` operation to a memory block not present in the cache results in a **write miss**.This memory block is brought to the cache.

6.4 Check for Bus Wait/Stalling of Cores

Table 5: Check for Bus Wait/Stalling of Cores

	Core0	**Core1**	**Core2**	**Core3**
Instruction	W 0x1000	W 0x1001	W 0x1002	W 0x1003
Idle Cycles	0	101	302	503
Execution Cycles	101	201	201	201
Miss Rate	100%	100%	100%	100%

- **Core 0 - Instruction:** A `Write` to address `0x1000` results in a **write miss**, as the block is not present in the cache. Since Core 0 is the first to issue a memory request, it does not experience any bus wait. The block is fetched, and execution proceeds without stalling.
- **Core 1 - Instruction:** A `Write` to address `0x1001` also causes a **write miss**. However, Core 1 must wait for Core 0's memory operation to complete. This results in **101 idle cycles** before Core 1 can proceed with execution.
- **Core 2 - Instruction:** A `Write` to address `0x1002` causes a **write miss** and must wait for both Core 0 and Core 1 to complete their memory operations, leading to **302 idle cycles**.
- **Core 3 - Instruction:** A `Write` to address `0x1003` also results in a **write miss** and must wait for Cores 0, 1, and 2 to finish. This causes **503 idle cycles**.

6.5 BONUS: False Sharing

6.5.1 False Sharing Testcase 1

Table 6: False Sharing 1

	Core0	**Core1**	**Core2**	**Core3**
Instruction 1	R 0x1000	W 0x1001	W 0x1002	W 0x1003
Instruction 2	R 0x1000			
Instruction 3	R 0x1000			
Idle Cycles	100	101	319	420
Execution Cycles	219	101	101	201
Miss Rate	66.67%	100%	100%	100%

- **Core 0 - Instruction 1:** A `Read` operation to a memory block not present in the cache results in a **read miss**. The memory block is fetched from memory and brought into the cache.
- **Core 0 - Instruction 2:** A `Read` operation to the same memory block now results in a **read hit**, as it is already present in the cache.

- **Core 1 - Instruction 1:** A `Write` operation to the same memory block results in a **write miss**. The block is fetched into Core 1's cache, and as a result, it invalidates the copy in Core 0's cache (due to the MESI protocol).
- **Core 0 - Instruction 3:** A subsequent `Read` operation to the same memory block now results in a **read miss**, as the block was invalidated by Core 1's write. This leads to the block being fetched again, demonstrating **false sharing**, which increases both miss rate and execution time.
- **Core 2 - Instruction 1:** A `Write` operation to a new memory block not present in the cache results in a **write miss**. The memory block is fetched into Core 2's cache.
- **Core 3 - Instruction 1:** A `Write` operation to another new memory block also results in a **write miss**, bringing the block into Core 3's cache.

6.5.2 False Sharing Testcase 2

Table 7: False Sharing 2

	Core0	**Core1**	**Core2**	**Core3**
Instruction 1	W 0x9000	W 0x9001	W 0x9002	W 0x9003
Instruction 2	R 0x9000	W 0x9001	W 0x9002	
Instruction 3	W 0x9003	R 0x9003		
Instruction 4		W 0x9003		
Idle Cycles	200	502	905	1307
Execution Cycles	303	604	402	201
Miss Rate	66.67%	75%	100%	100%

- **Core 0 - Instruction 1:** A `Write` operation to a memory block not present in the cache results in a **write miss**. The memory block is fetched from memory into the cache.
- **Core 0 - Instruction 2:** A `Read` operation to the same memory block results in a **read hit**, as the block is already present in the cache.
- **Core 1 - Instruction 1:** A `Write` operation to the same memory block results in a **write miss**. The block is fetched into Core 1's cache, invalidating Core 0's cache copy (due to MESI protocol).
- **Core 0 - Instruction 3:** A subsequent `Write` to the same block results in a **write miss**, as it was invalidated by Core 1. The block is fetched again into Core 0's cache and Core 1's copy is invalidated. This is an example of **false sharing**, which increases the miss rate and execution time.
- **Core 1 - Instruction 2:** Another `Write` to the same block results in a **write miss**, as it was invalidated by Core 0. The block is fetched again, and Core 0's copy is invalidated.
- **Core 1 - Instruction 3:** A `Read` to the same block results in a **read hit**, as the block is currently valid in Core 1's cache.
- **Core 2 - Instruction 1:** A `Write` to a new memory block (0x9002) results in a **write miss**. The block is fetched into Core 2's cache. If the block overlaps with one in Core 1, it may invalidate Core 1's copy.
- **Core 1 - Instruction 4:** A `Write` to the same block results in a **write miss** due to invalidation by Core 2. Core 1 fetches it again and invalidates Core 2's copy.
- **Core 2 - Instruction 2:** A `Write` to the same block results in a **write miss** due to invalidation by Core 1.

- **Core 3 - Instruction 1:** A `Write` to a new memory block results in a **write miss**, and the block is brought into Core 3's cache.

What is False Sharing and its effects?

False sharing occurs when multiple cores access different variables that reside on the same cache line, and at least one of the accesses is a write. Even though the variables are unrelated, the cache coherence protocol treats the cache line as a unit, causing invalidations and misses unnecessarily. This increases cache misses, leads to excessive memory traffic,increases time and degrades performance significantly.

7 Execution Time vs Cache Parameters

NOTE: Our execution is deterministic hence in each simulation we would receive the same output in each run if we keep the parameters same. We ran experiments where we varied one parameter at a time (others fixed):

- Cache size: 128B, 256B, 512B, 1024B, 2048B, 4096B, 8192B, 16384B
- Associativity: 1-way, 2-way, 4-way, 8-way
- Block size: 2B, 4B, 8B, 16B, 32B, 64B, 128B, 256B

7.1 Experiment 1: Cache Size

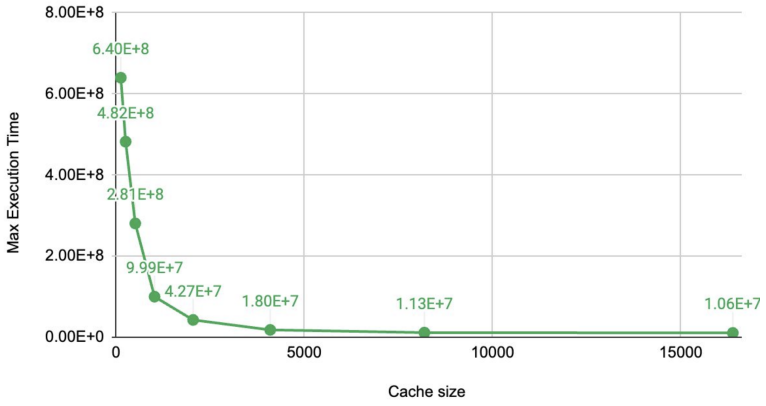


Figure 2: Max Execution time vs Cache Size (in Bytes)

Note: In order to vary the cache size, we kept the block size and associativity constant, and changed only the number of sets.

Formula:

$$\text{Cache Size} = \text{Number of Sets} \times \text{Block Size (32B)} \times \text{Associativity (Number of Ways} = 4)$$

Observation: Increasing the cache size reduces execution time exponentially by reducing miss rates and improving performance significantly up to a certain point. However, we observe diminishing returns in performance gains beyond 4 KB of cache size.

7.2 Experiment 2: Associativity

Observation: Increaing associativity reduces execution time by improving conflict miss handling but gains are modest beyond 2-way.

7.3 Experiment 3: Block Size

Observation: Larger blocks reduces execution time by reducing misses. However, too large blocks can waste cache space.

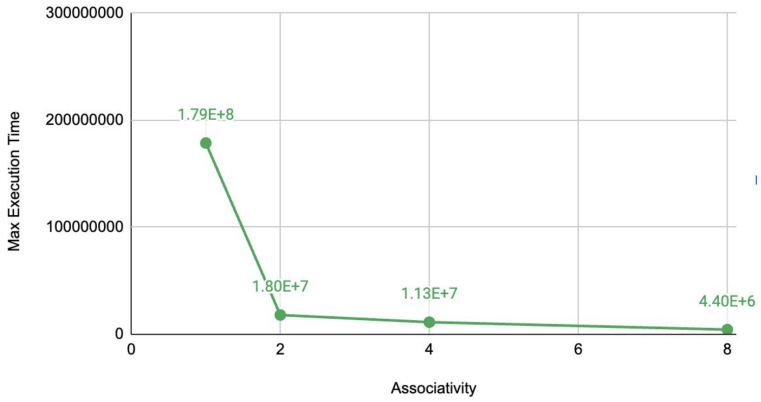


Figure 3: Max Execution Time vs Associativity

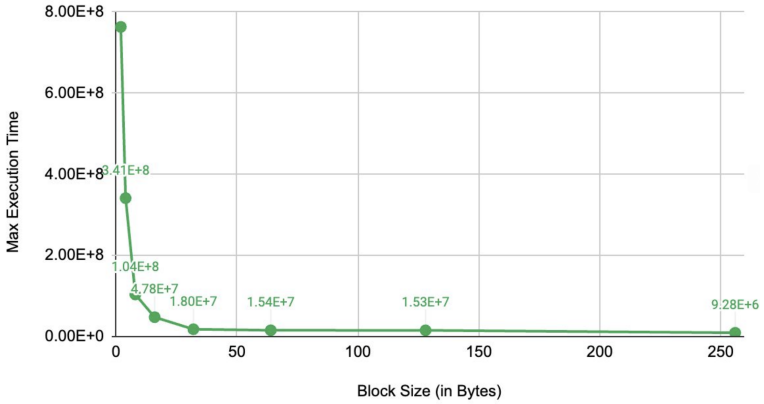


Figure 4: MAX Execution time vs Block Size (in Bytes)

Observation Highlights:

- Higher cache size leads to reduced capacity misses.
- Higher associativity leads to reduced conflict misses.
- The larger block size leads to reduced execution time and capacity misses initially but later on it leads to increased false sharing and wasting of Cache Space.

8 Conclusion

This project successfully develops a cycle-accurate simulator for L1 caches in a quad-core processor system, implementing the MESI cache coherence protocol. The goal was to gain a deeper understanding of real-world hardware-level cache coherence mechanisms by replicating their behavior in a controlled simulation environment. The simulator accurately models realistic delays, set-associative cache structures, and coordinated bus transactions to preserve coherence across cores.

- **MESI Protocol Implementation:** Each core maintains its own cache with lines in one of the four MESI states (Modified, Exclusive, Shared, Invalid). State transitions are faithfully modeled based on read/write operations and bus snooping events.
- **Cache Architecture Emulation:** Each L1 cache is independently configured with customizable block size, associativity, and total capacity. Efficient LRU replacement policies and constant-time tag lookups using maps emulate realistic cache behavior.

- **Coherent Bus Coordination:** A central bus manages inter-cache communication, handling invalidations, upgrades, and memory fetches. It enforces mutual exclusion and introduces cycle-level transaction delays to reflect timing bottlenecks.
- **Modular and Extendable Design:** The modular architecture makes the simulator easy to extend—for instance, to incorporate L2 cache support, implement alternate coherence protocols such as MOESI or MESIF, or simulate more complex memory hierarchies and execution models.
- **Deterministic Behavior:** The simulator yields deterministic outputs for identical input traces and configurations, allowing for reliable observation, debugging, and validation of results.
- **Performance Analysis:** Experimenting with different cache configurations (e.g., block size, associativity, total size) provided insights into performance trade-offs, highlighting latency sources and coherence overhead in multi-core systems.

In summary, the simulator serves as a valuable tool for exploring the principles of cache coherence in multicore architectures. It strikes a balance between accuracy and efficiency, and its modularity makes it suitable for future research, experimentation, or educational use.