COL 216 – Assignment 3

Anirudha Saraf 2023CS10806
Ayush Kumar Singh 2023CS10477

April 30, 2025

1 Implementation Details

Our implementation features a multi-core cache simulator with MESI protocol for coherence. The key components are:

- **MESI Protocol Implementation:** We have implemented the MESI (Modified, Exclusive, Shared, Invalid) protocol using an enum class to represent the four states. Each cache line tracks its current state and transitions according to protocol rules.
- **Cache Structure:** The Cache class implements a set-associative cache with parameterized geometry. The key cache parameters are:
  - `s`: Number of set index bits (determines number of sets S = 2^s)
  - `E`: Associativity (number of cache lines per set)
  - `b`: Number of block offset bits (determines block size B = 2^b bytes)
  - `t`: Number of tag bits (calculated as t = 32 - s - b for 32-bit addresses)

  Each cache line maintains tag, MESI state, data array of size B, and lastUsed timestamp for LRU replacement policy.
- **Bus Operations:** The simulator supports several bus operations:
  - BUSRD: Bus Read for read misses
  - BUSRDX: Bus Read Exclusive for write misses

  - BUSUPGR: Bus Upgrade for write hits to shared blocks

- **Snooping Mechanism:** Each cache monitors the bus for operations by other caches and responds with appropriate snoop results: NONE, SHARED, MODIFIED, or EXCLUSIVE.

- **Bus Transaction Tracking:** We explicitly track bus transactions with a dedicated counter that increments whenever the bus is used for coherence operations. Any data transferred on the bus from point A to point B contributes a bus transaction

1.1 Assumptions

- **Blocking Caches:** On a cache miss, the core stalls until the miss is resolved, but snoop operations still proceed.

- **Timing Assumptions:**
  - Read/Write to memory: 100 cycles in our implementation (reflected in code as `cycles += 100`)
  - Cache-to-cache transfer: 2 cycles per word (4 bytes), implemented as `cycles += 2 * wordsInBlock`
  - Read Hit: 1 cycle (No bus transactions), implemented as `cycles += 1`
  - Read Miss: 1 cycle + transfer time + memory access time if needed
  - Write Hit to Modified/Exclusive/Shared block: 1 cycle (no bus needed)
  - Invalidation takes 0 cycles (asynchronous process)
  - Lookup + Bus Signal takes 1 cycle.
  - Write Miss: 1 cycle + bus transaction + memory access time

- **Total Execution Cycles:** Tracked per cache using the `cyclesCount` field, which accumulates the time for all cache operations including penalties for misses, writebacks, and coherence transactions.

- **Idle Cycles:** Tracked using `idleCycles++` when a core needs the bus but cannot acquire it because another core is using it.

- **Cache Misses:** Incremented in the `processMemoryOperation` method when `hit` is false using `missCount++`. The miss rate is calculated as `missCount/totalInstructions × 100%`.
- **Cache Evictions:** Incremented when we need to replace a valid line in `findLRULine` using `evictionCount++`.
- **Writebacks:** Incremented in the following scenarios:
  - When evicting a modified line: `writebackCount++`
  - When another cache forces a modified line to be written back to memory: `writebackCount++`
  - When a modified line is downgraded to shared state: `writebackCount++`
- **Bus Invalidations:** Tracked using `busInvalidations++` when a core sends invalidation messages to other caches during a write operation.
- **Bus Data Traffic:** Tracked using `busDataTraffic += size` to count bytes transferred over the bus for both memory accesses and cache-to-cache transfers.
- **Bus Transactions:** Incremented using `incrementBusTransactions()` whenever a core uses the bus for any coherence operation, such as read misses, write operations requiring invalidations, and writebacks of modified data.
- **Statistical Tracking:**
  - Read Instructions: Number of read operations (`readCount`)
  - Write Instructions: Number of write operations (`writeCount`)
  - Total Instructions: Sum of reads and writes
  - Execution Cycles: Total cycles including active and idle time (`cyclesCount`)
  - Cache Miss Rate: `missCount/totalInstructions`

1.2 Main Classes and Data Structures

Cache Class:

- Fields:
  - `S`: Number of sets (derived as $S = 2^s$)
  - `E`: Associativity (number of cache lines per set)
  - `B`: Block size in bytes (derived as $B = 2^b$)
  - `s`: Number of set index bits
  - `b`: Number of block offset bits
  - `t`: Number of tag bits (calculated as t = 32 - s - b)
  - `sets`: Vector of cache sets, each containing multiple cache lines
  - Statistics: `readCount, writeCount, missCount, hitCount, evictionCount, writebackCount, cyclesCount, idleCycles, busInvalidations, busDataTraffic, busTransactions`
- Key Methods:
  - `parseAddress(address, tag, setIndex, blockOffset)`: Extracts address components
  - `isHit(address, cycle)`: Checks if address is in cache
  - `findLRULine(setIndex)`: Selects a victim line for replacement
  - `processMemoryOperation(address, op, otherCaches, cycles)`: Handles read/write operations
  - `handleBusOperation(address, busOp, requestorId)`: Responds to snooping requests
  - `incrementBusTransactions()`: Records bus usage

CacheSimulator Class:

- Fields:
  - `caches`: Vector of Cache objects for multiple cores
  - `traces`: Memory traces for each core
  - `s, E, b`: Cache configuration parameters

  - `busOwner, busFreeAtCycle`: Bus arbitration state

- Key Methods:
  - `loadTraces(baseFileName)`: Loads memory traces from files
  - `runSimulation()`: Executes the simulation
  - `printResults(out)`: Outputs statistics

1.3 Code Snippets

Listing 1: Finding an LRU Victim in Our Implementation

```
// Find LRU line for replacement
[[nodiscard]] int findLRULine(const uint32_t setIndex) const {
    const CacheSet& set = sets[setIndex];
    int lru_line = 0;
    uint64_t lruTime = std::numeric_limits<uint64_t>::max();

    for (int i = 0; i < E; i++) {
        if (set.lines[i].state == MESIState::INVALID) {
            return i; // Empty line found
        }
        if (set.lines[i].lastUsed < lruTime) {
            lruTime = set.lines[i].lastUsed;
            lru_line = i;
        }
    }
    return lru_line;
}
```

Listing 2: Bus Transaction Handling in runSimulation

```
// This core gets the bus
busOwner = coreId;

auto [usedBus, busOp] = caches[coreId].processMemoryOperation(
    address, op, otherCaches[coreId], cycles[coreId]);

if (usedBus) {
```

```
        // If the bus was used, increment the bus transactions counter
        caches[coreId].incrementBusTransactions();
    }

    // Handle snooping on other caches for block address
    const uint32_t blockAddress = caches[coreId].getBlockAddress(address);

    // Track invalidations performed by this operation
    int invalidationsDone = 0;

    for (Cache* otherCache : otherCaches[coreId]) {
        auto [result, wasInvalidated] = otherCache->handleBusOperation(block

        if (wasInvalidated) {
            invalidationsDone++;
        }
    }
```

1.4 Flowchart of Important Functions

Below is a high-level flow of the `CacheSimulator::runSimulation()` loop and `Cache::processMemoryOperation()` logic:

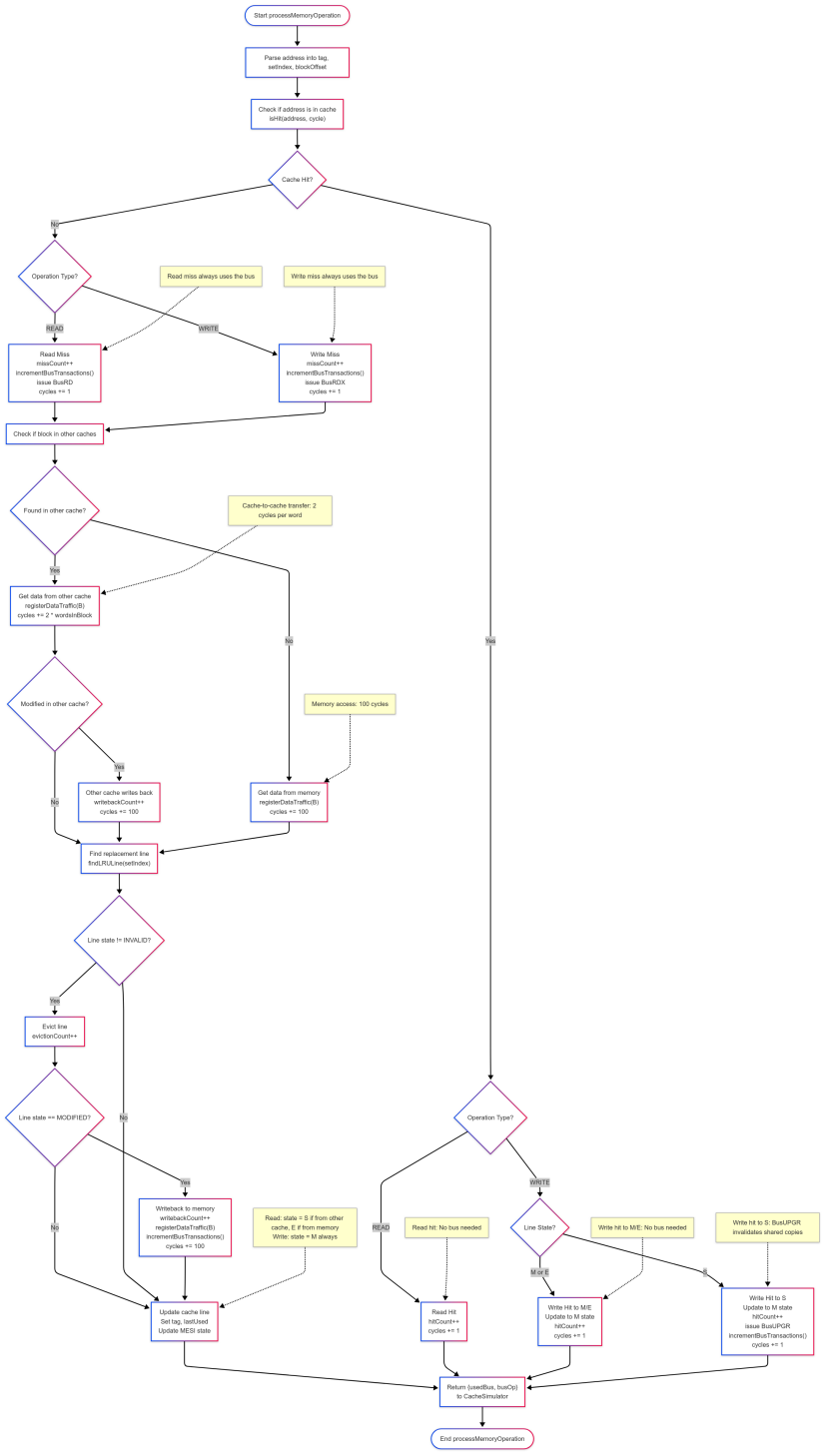


Figure 1: Cache Access Flowchart

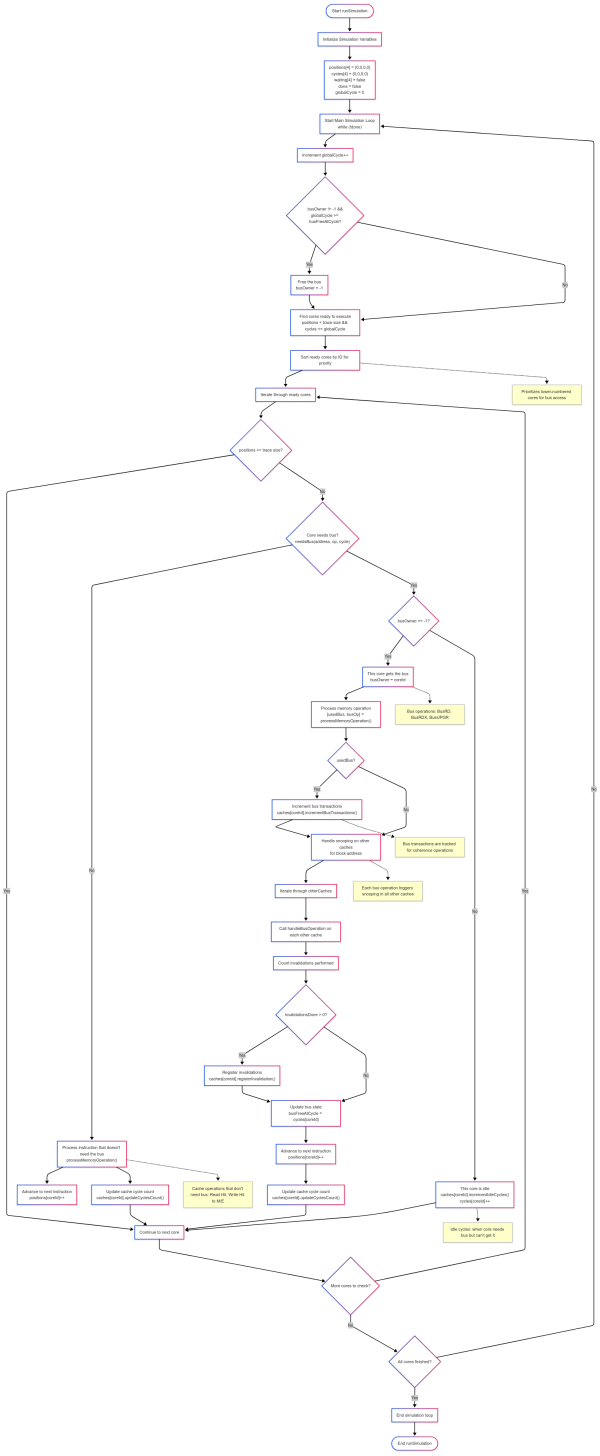


Figure 2: Simulator Run Flowchart

2 Experimental Evaluation

2.1 Setup

We configured our cache simulator with the following default L1 cache parameters:

- Sets: 64 sets ($s = 6$ bits, since $2^6 = 64$)
- Associativity: 2-way set associative ($E = 2$)
- Block size: 32 bytes ($b = 5$ bits, since $2^5 = 32$)
- Total cache size: $S \times E \times B = 64 \times 2 \times 32 = 4$ KB per core

We ran the simulator 10 times on fixed memory traces and recorded per-core execution cycles to evaluate performance consistency.

2.2 Arbitration Assumption

In our simulator, when multiple cores attempt to place transactions on the shared bus in the same cycle, priority is deterministically given to the lower-numbered core. This fixed arbitration scheme is implemented in the `runSimulation()` method where ready cores are sorted by ID and processed in order. This eliminates non-determinism in bus access order across simulation runs.

As a result of this deterministic bus arbitration, all simulator outputs (e.g., cache miss rates, execution cycles, idle cycles, and bus traffic) remain identical across all 10 runs for the same application trace. No variation is observed in any metric, as the simulation behavior is fully repeatable under fixed input and parameters.

3 Parameter Sensitivity Analysis

To understand how the cache parameters (s, E, b) affect performance, we conducted a sensitivity analysis by varying each parameter while keeping the others constant.

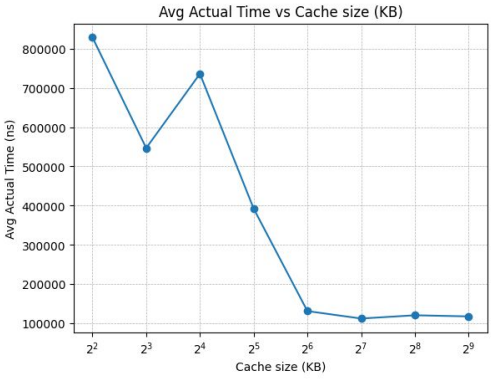
3.1 Maximum Execution Time Computation

We extended the simulator to record both simulated cycles and the actual wall-clock time (in nanoseconds) spent processing each instruction on every core. This is implemented through the `getMaxExecutionTime()` method which returns the maximum cycle count across all cores:

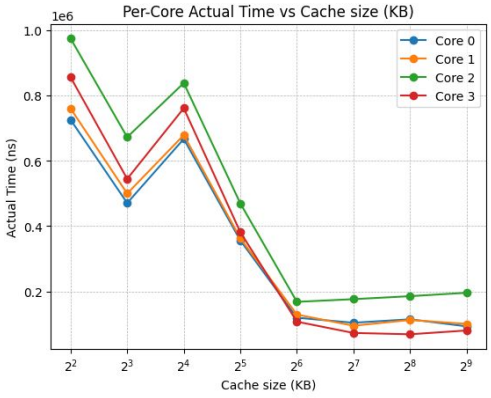
```
// Get maximum execution time (for experiments)
[[nodiscard]] uint64_t getMaxExecutionTime() const {
    uint64_t maxCycles = 0;
    for (int i = 0; i < 4; i++) {
        maxCycles = std::max(maxCycles, caches[i].getCyclesCount());
    }
    return maxCycles;
}
```

3.2 Varying Cache Size

We held associativity and block size at their default values (E = 4, b = 6 giving B = 64 bytes) and varied the number of set index bits (s) to achieve different cache sizes. By increasing s from 3 bits (8 sets) to 13 bits (8192 sets), we varied the total cache size from 8 KB ($2^3 \times 4 \times 2^6 = 2^{11}$ bytes) up to 1024 KB ($2^{13} \times 4 \times 2^6 = 2^{21}$ bytes).



(a) Avg actual time vs. cache size (KB).



(b) Per-core actual time vs. cache size (KB).

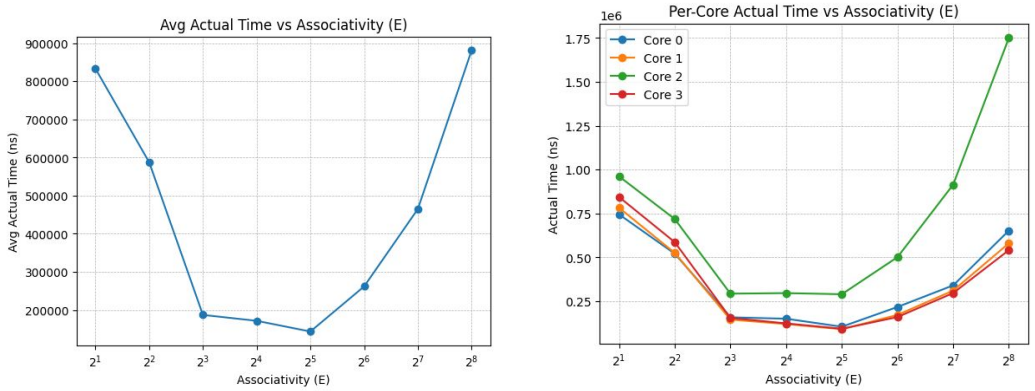
Figure 3: Effect of cache size on average and per-core execution times.

Observation: As cache size increases from 16 KB to 64 KB, the maximum execution time falls sharply. This corresponds to a drastic reduction in compulsory and capacity misses as the working set begins to fit within the cache. Beyond 64 KB the curve flattens, indicating that most remaining misses are either long-latency coherence or write-back events, or that the hot instruction/data footprint already resides in the L1/L2 hierarchy.

3.3 Varying Associativity

Next, we fixed the total cache size at 256 KB and block size at 64 bytes (b = 6), and varied the associativity parameter E from 2-way up to 256-way. As we increased E, we correspondingly decreased the number of sets (by reducing s) to maintain the same total cache size. For example:

- For E = 2: s = 12 (4096 sets) $\rightarrow$ 256 KB = $2^{12} \times 2 \times 2^6$ bytes
- For E = 4: s = 11 (2048 sets) $\rightarrow$ 256 KB = $2^{11} \times 4 \times 2^6$ bytes
- For E = 8: s = 10 (1024 sets) $\rightarrow$ 256 KB = $2^{10} \times 8 \times 2^6$ bytes



(a) Avg actual time vs. associativity (E).

(b) Per-core actual time vs. associativity (E).

Figure 4: Effect of associativity on (a) average and (b) per-core execution time.

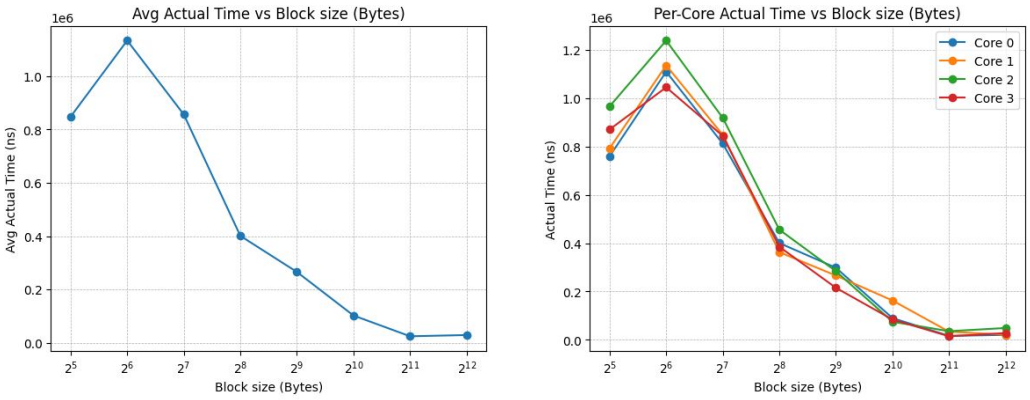
Observation: Increasing associativity steadily reduces conflict misses, but each additional way adds tag-lookup overhead and increases access latency. Initially (1$\rightarrow$4 ways) the drop in conflict misses yields a net gain, but

past 5 ways the diminishing returns in miss reduction are outweighed by the rising hit-latency penalty, causing the maximum time to plateau and even climb steeply.

3.4 Varying Block Size

Finally, with total cache size fixed at 256 KB and associativity at 4-way (E = 4), we varied the block size by adjusting the block offset bits (b) from 5 bits (32 bytes) up to 10 bits (1024 bytes). To maintain the same total cache size, we adjusted the number of sets (by changing s) accordingly:

- For b = 5 (32B blocks): s = 12 (4096 sets) $\rightarrow$ 256 KB = $2^{12} \times 4 \times 2^5$ bytes
- For b = 6 (64B blocks): s = 11 (2048 sets) $\rightarrow$ 256 KB = $2^{11} \times 4 \times 2^6$ bytes
- For b = 10 (1024B blocks): s = 7 (128 sets) $\rightarrow$ 256 KB = $2^7 \times 4 \times 2^{10}$ bytes



(a) Avg actual time vs. block size (Bytes).

(b) Per-core actual time vs. block size (Bytes).

Figure 5: Effect of block size on average and per-core execution times.

Observation: Very small blocks (32 B) incur high miss rates and thus long fetch latencies; very large blocks (512 B) suffer from excessive transfer times per miss and cache pollution. The "sweet spot" around 64 B balances

spatial locality against transfer overhead, minimizing the worst-case core execution time.