COL 216 Assignment 3: L1 Cache Simulator

Aaditya Sharma (2023CS10420)
Ishan Rehal (2023CS10019)

April 30, 2025

1 Implementation

1.1 Class Overview

The simulator is structured into five core components:

- **TraceParser**: Parses R/W traces into `Instruction` vectors.
- **Processor**: Simulates per-core execution and interfaces with its cache.
- **Cache**: Models an E-way set-associative L1 cache with 2^s sets and block size 2^b bytes, using write-back/write-allocate policy, LRU replacement, and MESI coherence.
- **Bus**: Central snooping bus managing coherence transactions.
- **StatsPrinter**: Gathers and prints simulation statistics.

1.2 Detailed Class Descriptions

- `TraceParser`: Reads each trace file and converts lines of the form `R/W 0xADDR` into a vector of `Instruction` structs, each with an operation type (READ/WRITE) and 32-bit address; uses C++ streams and handles I/O errors.
- **Processor**: Represents a core that issues at most one memory-access instruction per cycle, interacting with its private L1 cache and the shared bus—and stalling on pending cache misses.
  - Instruction vector and program counter (`currentInstructionIndex`).
  - Counters for total cycles, idle cycles, read count, write count.
  - Pointers to its L1 cache and the shared `Bus`.

  The `executeCycle()` method checks for a pending cache transaction to stall and increment idle cycles; otherwise it issues `cache.read()` or `cache.write()`, updates counters, and advances the PC.
- `Cache`: Implements an E-way set-associative cache with 2^s sets and block size 2^b bytes, write-back/write-allocate policy, LRU via per-line `lruCounter`, and MESI states stored in `meta[set][way]`.
  - `read(address,...)`: on hit updates LRU (1 cycle); on miss enqueues BusRd and marks pending.
  - `write(address,...)`: on Shared→Modified issues BusUpgr; on miss issues BusRdWITWr.

  - `resolvePendingTransaction()`: installs block after bus delay, evicts via LRU (handles writebacks), updates MESI state and traffic counters.
  - `handleBusTransaction(tx)`: snoops BusRd/BusRdX/BusUpgr to downgrade or invalidate lines, updating invalidation/writeback stats.

- `Bus`: Central snooping bus with queues for normal transactions and upgrade transactions. `resolveTransactions(caches)` first processes all BusUpgr invalidations, then the head normal transaction by broadcasting to caches, determining cache-to-cache vs. memory delay, and resolving the pending transaction in the requesting cache.

- `StatsPrinter`: Computes derived parameters (#sets, block size, cache size) and prints:
  - Simulation parameters summary.
  - Per-core statistics: total instructions, reads, writes, execution cycles, idle cycles, misses, miss rates, evictions, writebacks, invalidations, data traffic.
  - Bus summary: total bus transactions and total bytes transferred.

1.3 Data Structures

- **CacheLineMeta**: `bool valid`, `bool dirty`, `MESIState state`, `int lruCounter`.
- **TagArray**: 2D array `tags[set][way]` holding tag bits per cache line.
- **DataArray**: stores data words per block (used for traffic accounting).
- **BusTransaction**: POD struct with transaction type enum, address, source processor ID.
- **Instruction**: POD struct:

```
struct Instruction {
    OperationType op;    // READ or WRITE
    uint32_t     address; // 32-bit address
};
```

1.4 Control Flow Diagram

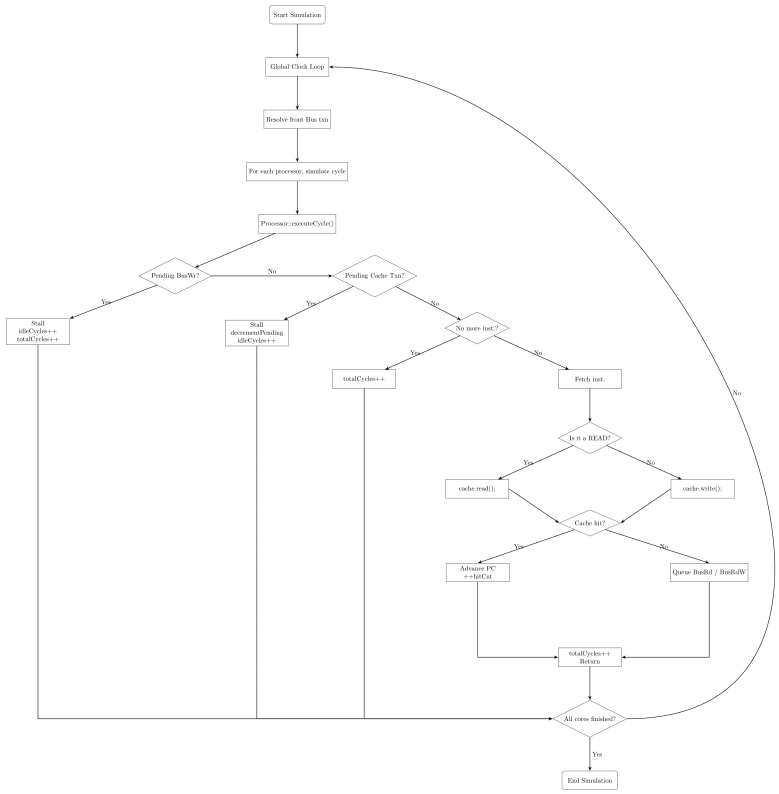

Figure 1: Global simulation control flow.

2 Design Decisions

In designing our simulator, we made several key choices to balance accuracy, clarity, and performance:

- **Micro-benchmark validation.** Before running full application traces, we exercised every cache and coherence corner case on small, hand-crafted inputs (2–5 operations). This allowed us to verify correct handling of hits, misses, writebacks, and invalidations in isolation.
- **MESI write-back and state transitions.**
  - When a cache line in `Modified` state is requested by another core, the owner immediately writes it back to memory (100cycles) and transitions to `Shared` along with a delay of 2N for cache to cache transfer.
  - If a core issues a write to a line currently `Modified` elsewhere, it incurs two 100cycle stalls: one for the remote writeback, and one for its own fetch (total 200cycles).
- **Bus transaction prioritization and instantaneous invalidation.** We maintain three priority queues on the bus:
  1. `BusUpgr` (invalidate-only) — highest priority. Invalidation messages are handled in a single cycle, allowing a core to upgrade a `Shared` line to `Modified` and others to `Invalidate` without delay once the bus grants the cycle.
  2. Write requests BusWr(for writing back to the memory)— next priority

3. Read requests (BusRd), Bus Read with the intent to Write(BusRdWITWr) — lowest priority.[both of them have same priority]

- **Deterministic core scheduling.** In each global cycle, we simulate Core 0, then Core 1, then Core 2, then Core 3 in strict order. This removes any non-determinism from bus arbitration (ties are broken by core index) and ensures identical results on every run for a given input.
- **Blocking cache model.** A core that misses on its L1 must stall until the miss resolves, but snoops and coherence actions continue to be processed. This accurately models in-flight coherence traffic while halting the issuing core.

3 Parameter Sensitivity Results

We varied one cache parameter at a time—doubling it from the default 64 sets, 2-way, 32 B configuration—and measured the maximum execution cycles across all cores for each application trace.

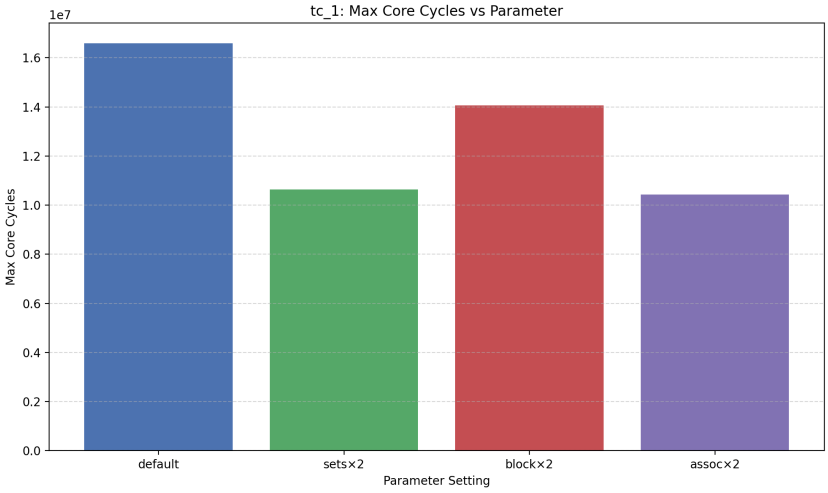

Figure 2: app1: Max execution cycles vs. parameter setting (default, sets×2, block×2, assoc×2).

Observation for app1 Doubling the number of sets (64 to 128) delivers the single largest performance win—max-core-cycles drop from about 16.6 M to 10.6 M (36% fewer cycles). This shows that capacity misses dominate in this workload.

Equally striking is the benefit of doubling associativity from 2-way to 4-way: max-cycles fall to about 10.4 M (37% fewer cycles), nearly matching the capacity boost. That tells us conflict misses were also a major contributor—hot addresses that kept evicting each other in a small-associativity cache now peacefully coexist across more ways, slashing stalls for both loads and writes.

By contrast, doubling the block size from 32 B to 64 B only reduces cycles to about 14.1 M (15% fewer). Larger blocks do indeed capture extra spatial locality—fetching neighboring words in one go—but they also double the miss penalty (2 N cycles to transfer a full cache line) and tie up the bus longer on each refill. The net effect is only a moderate gain, because the extra hits you pick up aren't enough to outweigh the heavier transfer cost on each miss.

In summary, for this trace, capacity and conflict misses are the dominant performance limiters—so increasing sets or ways yields dramatic speedups—whereas spatial locality is a secondary factor, giving only a modest improvement when blocks are enlarged.

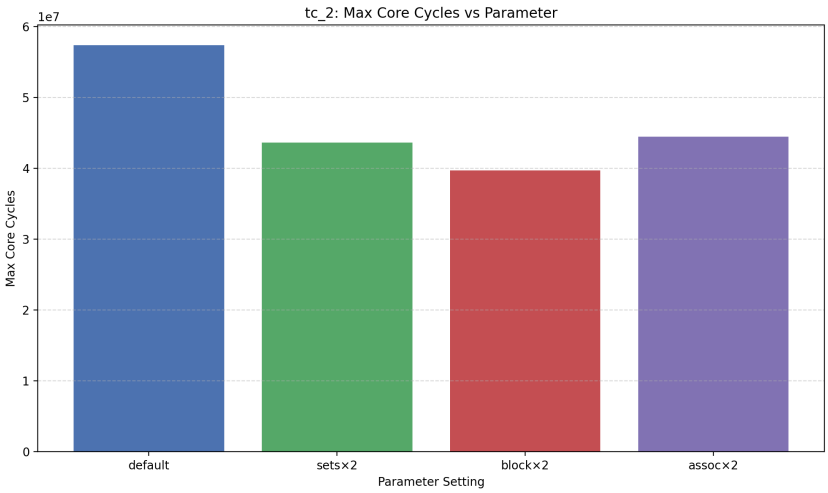

Figure 3: app2: Max execution cycles vs. parameter setting (default, sets×2, block×2, assoc×2).

Observation for app2 For tc 2 we see a very different sensitivity profile:

Default to Block×2: Max cycles drop from about 57 M to 40 M, a 30 % reduction—by far the largest improvement. This tells us that spatial locality rules in this workload: fetching 64 B at a time instead of 32 B dramatically cuts the number of misses, because the cores stream through long, contiguous data regions.

Default to Sets×2: Cycles fall from 57 M to 43 M, roughly a 25 % gain. Larger total capacity (8 KB) also helps by capturing a bigger working set, but its impact is secondary to block size.

Default to Assoc×2: Cycles drop to 44 M, about a 23 % improvement. Boosting from 2-way to 4-way reduces conflict misses, but since this trace already streams through data sequentially, there simply aren't as many hot lines competing in the same set.

In summary, tc 2's dominant performance limiter is spatial locality (hence block doubling gives the biggest win), followed by capacity, with conflict misses playing a smaller role. This contrasts with tc 1, where capacity and conflict were the biggest factors.

4 Conclusions

From our parameter-sensitivity study on both application traces, we draw the following key takeaways:

- **Universal benefit:** Doubling any single cache parameter (sets, block size, or associativity) yields a clear reduction in maximum core cycles, demonstrating that larger or more flexible caches always help reduce costly misses.
- **Workload-dependent impact:** The relative gain depends on the trace's access pattern—capacity-bound workloads see the biggest win from more sets, while streaming workloads benefit most from larger block sizes.
- **Conflict vs. capacity vs. spatial locality:**
  - More sets chiefly mitigates *capacity* misses by enlarging the total storage.
  - Larger blocks exploit *spatial locality* but incur higher per-miss transfer costs.
  - Higher associativity reduces *conflict* misses, with diminishing returns once capacity and spatial locality are satisfied.

- **Diminishing returns:** Beyond a certain point—once the working set largely fits and spatial streams are captured—further increases in any one parameter produce progressively smaller performance improvements.

5 Bonus: False Sharing

To illustrate the performance impact of false sharing, we crafted a micro-benchmark in which two cores repeatedly write to different 4-byte words that reside in the same 32-byte cache line, while the other two cores perform benign reads elsewhere. The traces are:

- `false_sharing_proc0.trace`: 20 writes to 0x1000
- `false_sharing_proc1.trace`: 20 writes to 0x1004
- `false_sharing_proc2.trace`: 20 reads of 0x2000
- `false_sharing_proc3.trace`: 20 reads of 0x2000

5.1 Simulation Output

```
Simulation Parameters:
  Set Index Bits: 6, E = 2, b = 5 (32B blocks), 64 sets, 4KB/cache

Core 0 Statistics:
  Total Instr: 20, Writes: 20
  Cycles: 640, Idle: 618
  Misses: 2 (10%), Evictions: 1, Writebacks: 1
  Invalidations: 1, Traffic: 96B

Core 1 Statistics:
  Total Instr: 20, Writes: 20
  Cycles: 621, Idle: 401
  Misses: 1 (5%), Evictions: 0, Writebacks: 1
  Invalidations: 1, Traffic: 64B

Core 2 / Core 3 (reads):
  Cycles 621, Idle 410, Misses: 1 each, Evictions: 1/0, Traffic: 32B each

Overall Bus:
  Transactions: 7, Traffic: 224B
  Global Clock: 641 cycles
```

5.2 Analysis of False Sharing

- **Same-line writes trigger constant invalidations.** Cores 0 and 1 write to addresses 0x1000 and 0x1004, which lie in the same 32B block. Every write from one core invalidates the other core's copy, forcing a bus transaction and a cache miss on the next write by the other core.
- **High idle-cycle ratios.** Core 0's idle cycles (618/640 97%) and Core 1's idle (401/621 65%) reflect that most cycles are spent waiting on cache misses and coherence traffic rather than useful work.

- **Frequent bus transactions and traffic.** Seven bus transactions for only 40 memory operations (0.175 transactions per access) and 224B of coherence traffic (vs. only 80B of actual data read/written) demonstrate the overhead of false sharing.
- **Evictions and writebacks due to ping-pong.** Core 0 incurs one eviction and one writeback, even though it writes the same word repeatedly—because each round of invalidation forces the line out and back into the cache.
- **Contrast with non-sharing case.** If Core 1 had written to 0x0000 instead, each core would see only a single miss to fetch the line, zero invalidations, and far fewer idle cycles and bus transactions.
- **Key takeaway:** False sharing—independent variables lying in the same cache block—can dominate cache performance, turning what would be low-latency local writes into high-latency, bus-bound operations.

5.3 Non-Sharing Variant

To isolate the effect of false sharing, we reran the same micro-benchmark but had Core 1 write to 0x0004 (a different 32 B line) instead of 0x1004. The rest of the setup (Core 0 on 0x1000, Cores 2/3 benign reads at 0x2000) remains unchanged.

```
Simulation Parameters:
  Set Index Bits: 6, E = 2, b = 5 (32B blocks), 64 sets, 4KB/cache

Core 0 Statistics:
  Total Instr: 20, Writes: 20
  Cycles: 320, Idle: 100
  Misses: 1 (5%), Evictions: 1, Writebacks: 0
  Invalidations: 0, Traffic: 32B

Core 1 Statistics:
  Total Instr: 20, Writes: 20
  Cycles: 320, Idle: 201
  Misses: 1 (5%), Evictions: 1, Writebacks: 0
  Invalidations: 0, Traffic: 32B

Core 2 / Core 3 (reads):
  Cycles 323/340, Idle 302/319
  Misses: 1 each, Evictions: 1/0, Writebacks: 0, Invalidations: 0
  Traffic: 32B each

Overall Bus:
  Transactions: 4, Traffic: 128B
  Global Clock: 341 cycles
```

Comparison to False-Sharing Case

- **Bus transactions:** Non-sharing only issues 4 vs. 7 in the false-sharing run (↓43%), because no ping-pong invalidations occur.
- **Idle cycles:** Core 0 idle drops from 618 → 100, Core 1 from 401 → 201, freeing up 75–85 % of cycles for useful work.

- **Invalidations and writebacks:** Zero invalidations/writebacks in the non-sharing case.
- **Traffic reduction:** Bus traffic is halved (128B vs. 224B), closely matching the reduction in bus transactions.

This demonstrates that simply placing independent variables in separate cache lines eliminates the coherence overhead of false sharing.

Repository

The full source code, Makefile, sample traces, and this report are available on GitHub:

https://github.com/ishan-rehal/COL216$_{A}ssignment_{3}$