COL 216 Assignment 3:
L1 Cache Simulator for Quad-Core Processor

Aditya Anand (2023CS50284)
Vansh Ramani (2023CS50804)

May 9, 2025

1 Implementation

1.1 Class Overview

The simulator comprises six key modules, each handling a specific aspect of the L1 cache simulation:

- **TraceReader**: Reads and parses memory access traces for each core.
- **Core**: Simulates a processor core's execution, issuing memory instructions and managing stalls.
- **Cache**: Models a private, E-way set-associative L1 data cache per core with 2^s sets and 2^b-byte blocks. Supports write-back, write-allocate, LRU replacement, and MESI coherence.
- **Bus**: A shared communication medium handling coherence transactions and arbitration.
- **Simulator**: Orchestrates core execution, bus activity, and simulation cycles.
- **StatsPrinter**: Aggregates and reports cache performance metrics.

1.2 Module Descriptions

1.2.1 TraceReader

Reads trace files (one per core) line-by-line. Each line specifies a memory operation (R or W) and a hexadecimal address, converted into `TraceEntry` structs with a 32-bit address and operation type. Uses standard C++ I/O with input validation.

1.2.2 Core

Represents a CPU core, executing one memory operation per cycle unless stalled. Communicates with its private L1 cache and other caches via the bus. Features:

- Instruction stream from the trace.
- Counters for instructions, read/write operations, idle cycles, and total cycles.
- State tracking for stalls due to cache misses or coherence waits.
- `tick()` function to issue instructions or increment idle time.

1.2.3 Cache

Each core has a local cache configured with:

- E lines per set, 2^s sets, and block size of 2^b bytes.
- Cache lines storing tags, MESI states, LRU timestamps, valid/dirty flags.
- Read/write methods updating LRU order on hits and initiating coherence on misses.
- `allocateBlock()` and `handleMiss()` for block allocation and evictions.
- Snooping support for BusRd, BusRdX, and MESI transitions (e.g., Shared $\rightarrow$ Invalid).

1.2.4 Bus

Manages inter-cache coherence communication:

- Round-robin arbitration for transactions.
- Queues for pending transactions, prioritizing BusRdX over BusRd or WriteBack.
- `tick()` processes transactions, handles snoop responses, and triggers cache state changes.
- Tracks bytes transferred and latency for memory vs. cache-to-cache transfers.

1.2.5 Simulator

Coordinates cores and caches. In each cycle:

- Calls `tick()` on each core.
- Updates the bus and processes snoop responses.
- Monitors termination when all cores finish their traces.
- Finalizes per-core cycle counts and statistics.

1.2.6 StatsPrinter

Reports:

- Cache parameters (size, associativity, block size).
- Per-core stats: accesses, execution/idle cycles, misses, evictions, writebacks, invalidations.
- Bus stats: total transactions and data transferred.

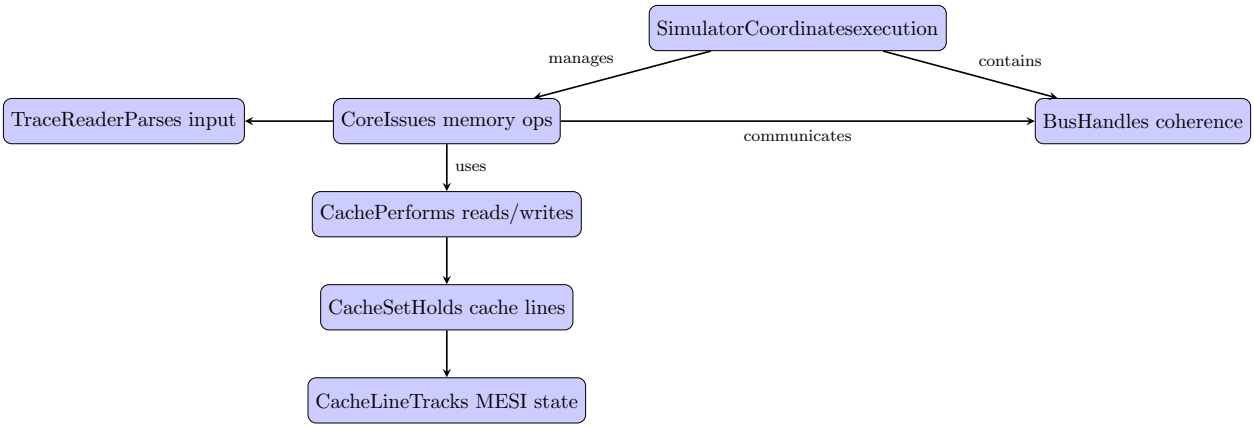

Figure 1: Class interaction diagram of the L1 Cache Simulator

2 Data Structures

Core data structures include:

- **CacheLine**: Stores `tag`, `flags.valid`, `flags.state` (MESI), `lastUsedCycle` for LRU.
- **CacheSet**: `std::vector<CacheLine>` for lines, $\texttt{std::unordered}_{m}ap for tag lookups.\texttt{MESI State}(Enum\texttt{Cach}$ $MODIFIED, EXCLUSIVE, SHARED, INVALID$.
- **BusRequestType** (Enum): BusRd, BusRdX, WriteBack.
- **BusTransaction**: `requesterId`, `address`, `type`, `startCycle`, `completionCycle`, `servedByCache`.
- **TraceEntry**: `op` (READ/WRITE), `addr` (32-bit).
- **Statistics**: Counters for hits, misses, evictions, writebacks, invalidations, idle cycles, instruction count, total cycles.

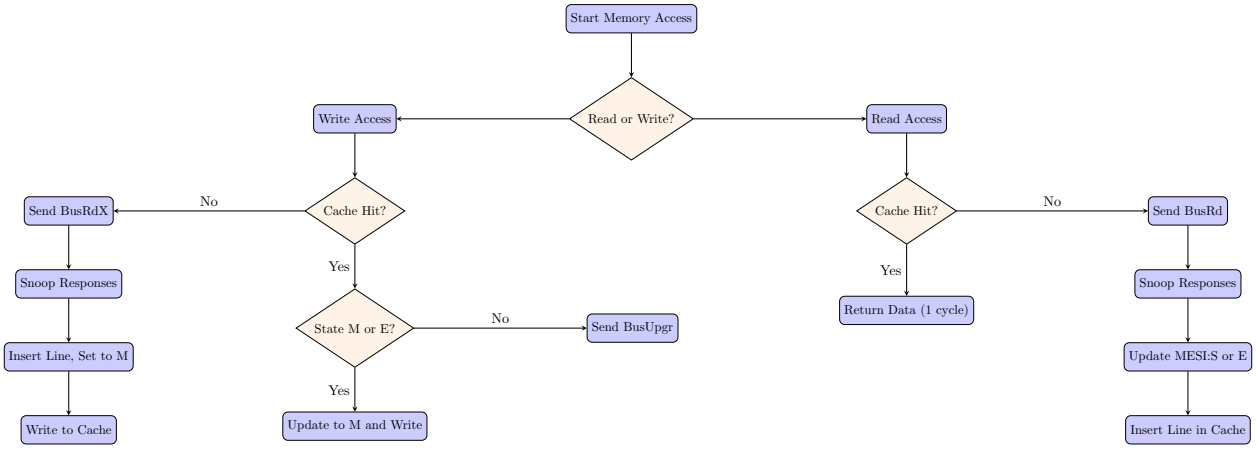

Figure 2: Memory access control flow for L1 cache simulator

3 Control Flow Diagram

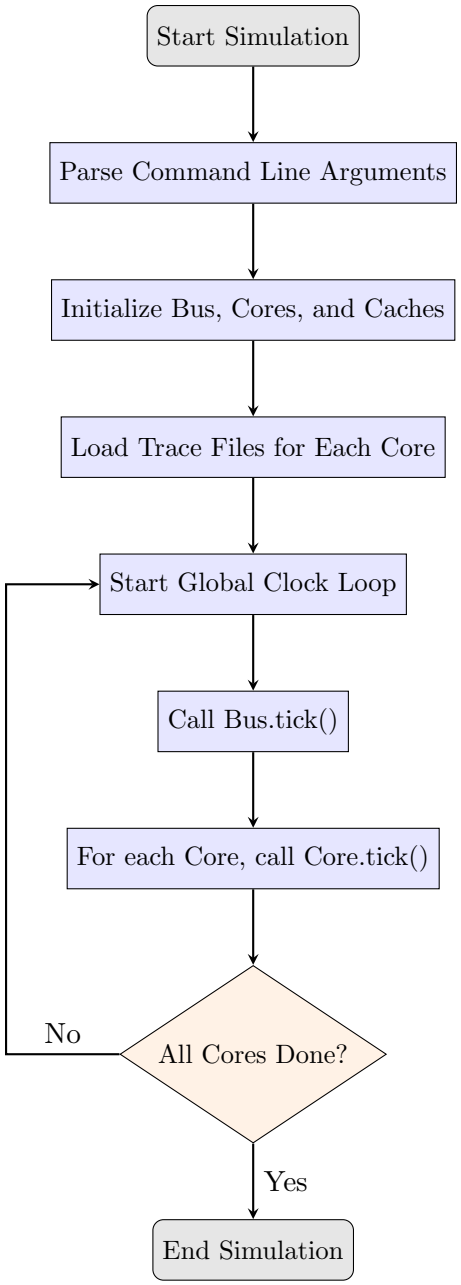

Figure 3: Top-level global simulation control flow for L1 cache simulator

4 Design Decisions

The L1 cache simulator incorporates key design choices for realistic behavior, deterministic execution, and manageable complexity:

1. **Idle Cycle Definition**: Idle cycles occur when a core waits for memory operations, including bus coherence or memory fetches. Total cycles = execution cycles + idle cycles.

2. **MESI Protocol**: Implements full MESI with transitions:
    - Shared → Modified: Issues BusRdX, triggers invalidations.
    - Modified → Shared/Invalid: Via snooping on BusRd/BusRdX.
    - Modified writebacks: 100-cycle latency.
3. **Writebacks**:
    - Eviction: Synchronous writeback of dirty blocks.
    - Snoop-triggered: Writeback before Modified → Shared/Invalid.
4. **Bus Arbitration**: One transaction at a time, prioritized:
    (a) BusUpgr (1 cycle).
    (b) BusRdX, WriteBack.
    (c) BusRd (lowest priority).

    Tie-breaking by ascending core ID.
5. **Deterministic Core Order**: Cores 0–3 tick in fixed order for reproducibility.
6. **Blocking Cache**: Core stalls on misses until transaction completes; caches respond to snoops in parallel.
7. **Memory Latencies**:
    - L1 hit: 1 cycle.
    - Memory fetch: 100 cycles.
    - Cache-to-cache: 2 cycles per 4-byte word.
    - Writeback: 100 cycles.
8. **Cache Initialization**: All lines invalid at start (cold cache).
9. **Word/Block Size**: 4-byte words, 2^b-byte blocks.
10. **Simplified Data**: Only tags, states, and timing modeled.
11. **No Bus Pipelining**: One transaction at a time.
12. **Microbenchmark Verification**: Validated MESI and writebacks with 2–5 instruction traces.

5 Performance Analysis

To evaluate cache parameter impacts, we conducted experiments varying one parameter at a time while keeping others fixed at:

- Number of sets: $2^s = 64$
- Associativity: $E = 2$
- Block size: $B = 32$ bytes

For each parameter, we doubled its value across runs and measured the **maximum execution cycles across all cores**, reflecting total workload completion time, including computation and stalls due to memory or coherence delays. Experiments used provided application traces, with results recorded for comparison.

6 Implementation Challenges

Key challenges encountered include:

1. **Precise Cycle Synchronization**: Managing a global simulation loop while allowing independent core instruction issuance required careful delay handling for memory fetches, write-backs, and coherence events.

2. **MESI State Coordination**: Ensuring correct MESI transitions across cores demanded rigorous tracking of local and remote operations, especially for Modified $\rightarrow$ Shared or Exclusive $\rightarrow$ Invalid transitions.

3. **Bus Arbitration and Fairness**: Modeling realistic contention with deterministic behavior involved priority-based selection and round-robin tie-breaking.

4. **Blocking Behavior and Stall Management**: Simulating blocking caches—halting cores on misses while responding to coherence requests—required separating execution from snoop handling and tracking core resumption.

5. **Edge Case Handling**: Malformed trace files, invalid inputs, and misaligned addresses necessitated robust validation.

6. **Comprehensive Metric Tracking**: Accurate tracking of idle cycles, evictions, writebacks, and data traffic required consistent counters across modules without double-counting.

7 Conclusion

Our L1 cache simulator accurately models a quad-core processor with MESI coherence, write-back/write-allocate policy, and LRU eviction. It captures private cache and shared memory interactions, providing cycle-accurate statistics.

Key trends from parameter variation experiments:

1. **Cache Size**: Larger caches reduce capacity and compulsory misses, but benefits plateau when the working set is fully accommodated.

2. **Associativity**: Higher associativity (2-way or 4-way) reduces conflict misses, improving hit rates but potentially increasing access latency.

3. **Block Size**: Balancing transfer time and spatial locality, block sizes of 32–64 bytes performed best for our traces.

These results highlight cache architecture trade-offs and the importance of workload-specific configurations. The simulator enables functional validation and performance exploration in multi-core systems.

8 Repository

Full source code and report:
https://github.com/VanshRamani/CacheSim