COL216 - Cache Simulator Report

Rommyarup Das, 2023CS50666
Shaurya Vardhan, 2023CS50220

April 30, 2025

1. Problem Statement

Simulate L1 cache for a quad-core processor system with MESI-based cache coherence, write-back, write-allocate, and LRU replacement policies. Each core processes a trace of 32-bit memory references in parallel, and caches communicate over a shared bus. Assumptions beyond the given specification are clearly stated where used.

1.1 Simulation Details

1. Memory addresses are 32-bit; shorter addresses are zero-extended at MSBs.
2. Data word size is 4 bytes per memory reference.
3. Only data caches (L1) are modeled; no instruction cache or L2.
4. Each core has a private L1 data cache backed by main memory.
5. L1 uses write-back, write-allocate, and LRU replacement.
6. Cache coherence is maintained via MESI protocol.
7. Caches start empty; bus ties are broken arbitrarily.
8. L1 hit latency: 1 cycle; memory fetch: +100 cycles; cache-to-cache transfer: 2 cycles/word; write-back eviction: 100 cycles.

9. Caches are blocking: a miss stalls the core but snoop events still proceed.
10. One memory instruction per core per cycle maximum.

1.2 Input Details and Command-Line Parameters

The simulator `L1simulate` accepts:

- `-t <appname>`: base name for four trace files (proc0.trace to proc3.trace).
- `-s <s>`: set index bits (cache sets = 2^s).
- `-E <E>`: associativity (lines per set).
- `-b <b>`: block offset bits (block size = 2^b bytes).
- `-o <outfile>`: output log file name for plotting.
- `-h`: help.

1.3 Simulation Output

Per run, the simulator reports for each core:

1. Read/write instruction counts.
2. Total execution cycles.
3. Idle (stall) cycles.
4. Data-cache miss rate (misses/accesses).
5. Cache evictions count.
6. Write-back count.
7. Bus invalidations.
8. Bus traffic (bytes).

2. Implementation

2.1 Main Classes and Data Structures

- **CacheLine**: tag, valid, dirty bits, MESI state.
- **CacheSet**: vector of CacheLine, LRU queue.
- **Cache**: array of CacheSet, `read()/write()` methods.
- **Processor**: encapsulates a Cache and processes its trace.
- **Simulator**: orchestrates cores, bus transactions, and collects statistics.

2.2 Core Simulation Workflow

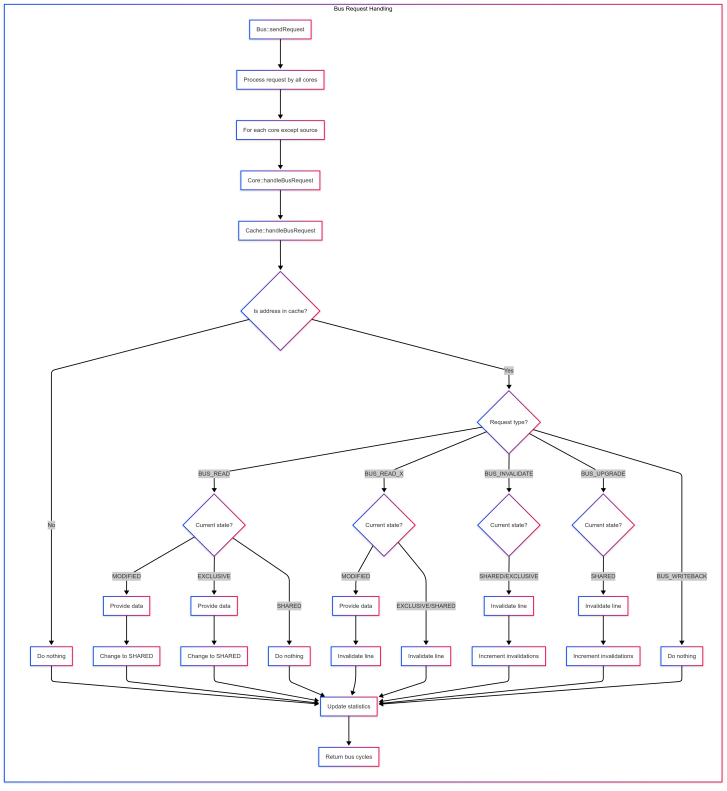

Figure 1: Bus Request Handling

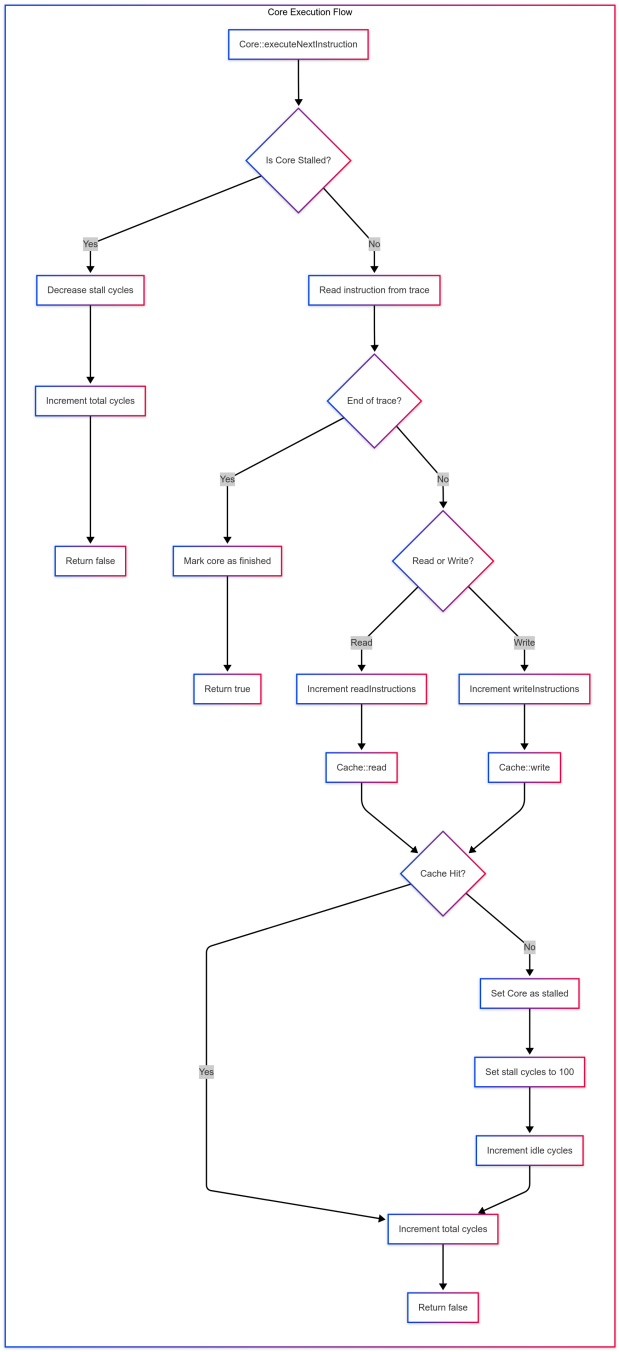

Figure 2: Core Execution Flow

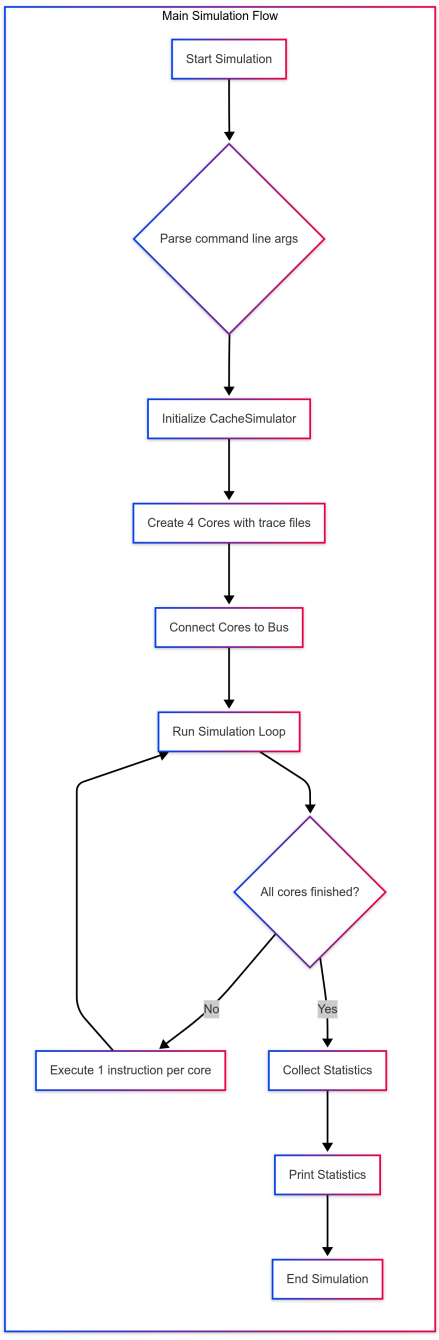

Figure 3: Main Simulation Flow

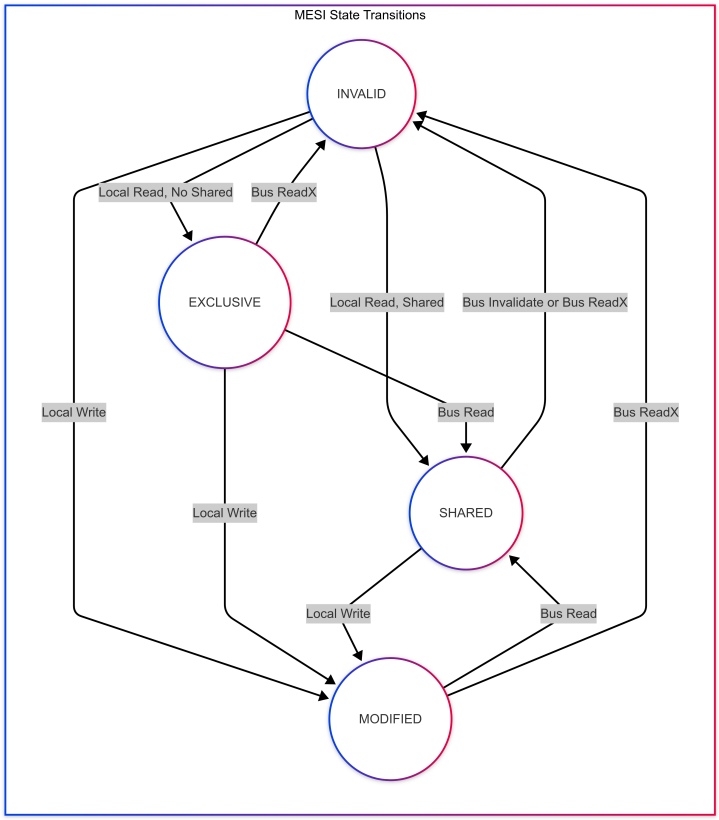

Figure 4: MESI State Transitions

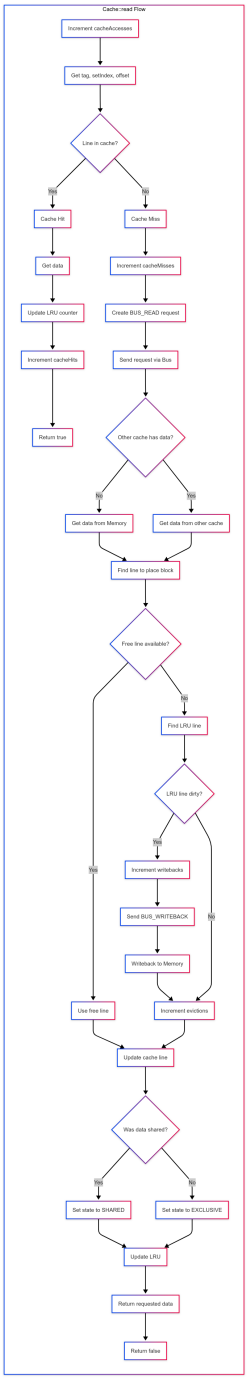

Figure 5: Cache Read Flow

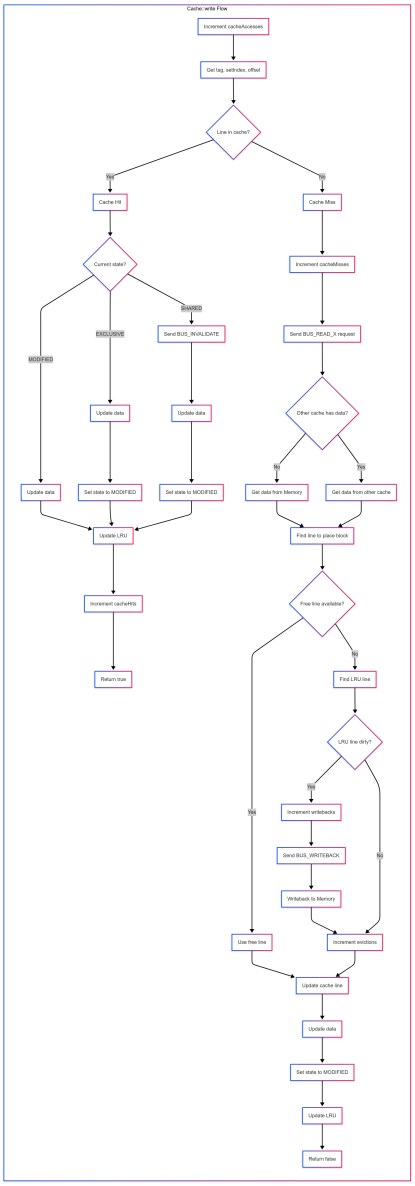

Figure 6: Cache Write Flow

3. Experimental Evaluation

Using default: 4KB cache ($s = 7$, $E = 2$, $b = 5$), run 10 trials on `app1`. Outputs are deterministic given traces, so distributions show zero variance

for cycle counts and miss rates; stochastic elements (e.g., tie-breaking) affect bus invalidations and traffic minimally.

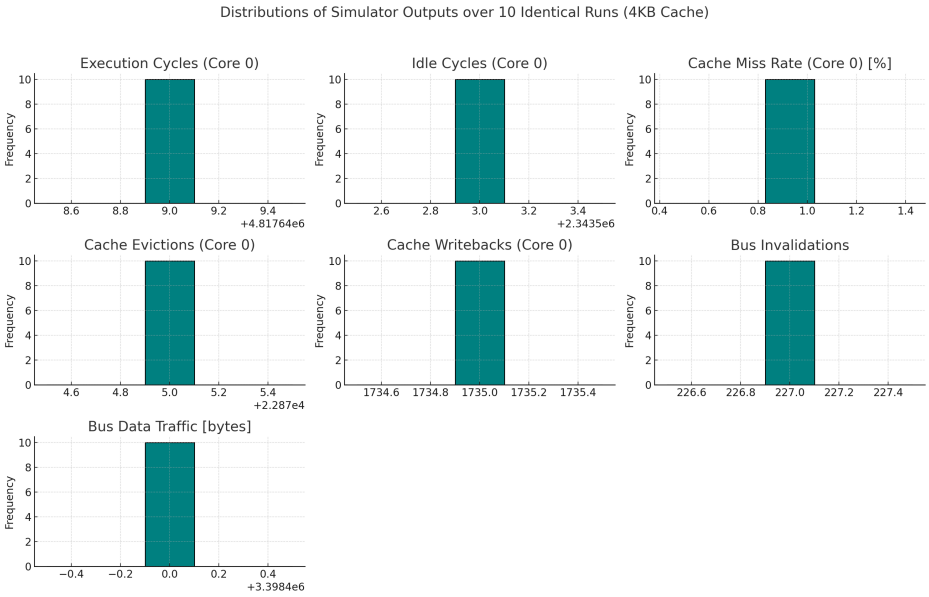

Figure 7: Distribution of total cycles over 10 runs

4. Parameter Sensitivity Analysis

We extended the simulator with a helper:

```
uint64_t getMaxExecutionTime() {
    uint64_t max_t = 0;
    for (auto &core : cores)
        max_t = max(max_t, core.total_cycles);
    return max_t;
}
```

Varying each parameter individually in powers of two:

- Cache size: 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1024KB.
- Associativity: 2, 4, 8, 16, 32, 64, 128, 256, 512.

- Block size: 16B, 32B, 64B, 128B, 256B, 512B, 1024B, 2048B, 4096B.

Figure 8: Max execution time vs. cache size

Observations: Increasing cache size reduces miss penalties and stalls; higher associativity lowers conflict misses initially but with diminishing returns; larger blocks improve spatial locality but can increase miss penalty.

5. Interesting Trace Analysis - Demonstrating False Sharing

Each file contains a single memory write to a different offset:

```
app4_proc0.trace:  W 0x2000
app4_proc1.trace:  W 0x2008
app4_proc2.trace:  W 0x2010
app4_proc3.trace:  W 0x2018
```

Each address is 8 bytes apart (0x8), so:

- All four addresses lie within a single 64-byte cache line (assuming block size b = 6, i.e., 64 bytes).

- The first address is 0x2000, and the last is 0x2018, so the entire line spans from 0x2000 to 0x203F.

This cause false sharing because all processors (0–3) are writing to different variables that share the same cache line:

- Suppose each processor has its own private L1 cache (a common setup).
- When proc0 writes to 0x2000, it loads the cache line [0x2000–0x203F] into its cache.
- Then proc1 writes to 0x2008 — that line must be loaded into proc1's cache and invalidated in proc0's.
- Then proc2 writes to 0x2010, evicting from proc1's cache.
- Then proc3 writes to 0x2018, invalidating from proc2's.

This ping-ponging means:

- Every write forces the line to migrate between cores
- Even though each core touches only its own 8-byte slice, the full cache line must move

So each write results in a cache miss due to coherence-induced invalidation.