L1 Cache Simulator

Sabhya Arora, Rachit Bhalani
2023CS11177, 2023CS10961
COL216: Computer Architecture
Assignment 3

April 30, 2025

Contents

1 Implementation Overview

1.1 Design

We implemented a cycle-accurate simulator in C++ for a quad-core processor system, where each processor has a private L1 data cache. The simulation supports standard cache coherence and memory hierarchy features aligned with modern shared-memory multiprocessors. Key components of the design include:

- **MESI Cache Coherence Protocol**: Each cache block can be in one of the four states: Modified, Exclusive, Shared, or Invalid. Transitions between states are governed by local processor operations (read/write hits/misses) and remote snooping on bus transactions (e.g., memory reads, RWITM, invalidates). Dirty (Modified) blocks are written back to memory upon eviction.
- **Write Policy**: The simulator uses a write-back, write-allocate policy. This means on a write miss, a cache block is fetched into the cache before writing, and updates are only propagated to memory when the block is evicted.
- **Bus Requests and Arbitration**: A central snooping bus is used for coherence management. All caches snoop the bus and react to bus transactions initiated by others. When multiple processors initiate bus transactions simultaneously, arbitration is performed using processor ID — the processor with the lowest ID wins.
- **LRU Replacement**: Each set in the cache tracks access timestamps for its ways. The Least Recently Used block (determined using minimum access time) is evicted on insert if the set is full. Access times are updated on every read/write hit and miss.
- **Memory Model and Timing**:
  - L1 cache hits take 1 cycle.
  - A block fetched from memory takes 100 cycles.
  - Transfers of blocks between caches take $2N$ cycles where N is the number of words per block (each word = 4 bytes).
  - Evicting a dirty block costs 100 cycles.
- **Blocking Caches**: Each processor halts during a cache miss until the cache line is fetched and installed. However, it continues to service coherence requests (snooping) from the bus.
- **Trace-Driven Simulation**: Each processor executes its own memory trace file consisting of read/write operations. The simulator supports the execution of 4 parallel traces representing a multithreaded program on a quad-core system.

The simulator was carefully designed to match the assignment specification: each memory access is 4 bytes, the address space is 32-bit padded, and the simulation ends when all instructions from all 4 cores complete execution.

1.2 Control Flow Diagram

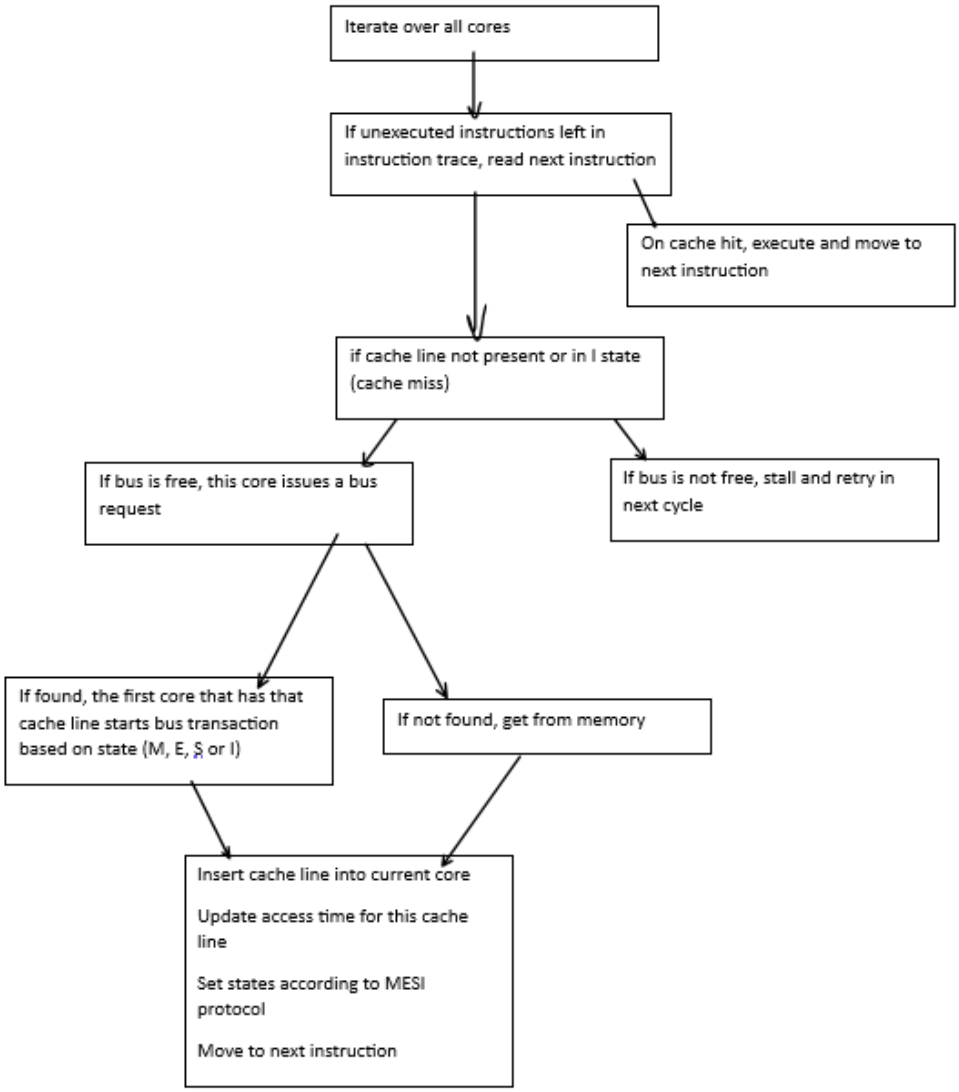

Figure 1: High-level control flow of the simulation loop.

1.3 Assumptions

- **Core and Cache Architecture:** The simulation models a quad-core processor system, with each core having a private, exclusive L1 data cache. There is no shared cache. Each L1 cache is organized as a set-associative cache with configurable parameters (number of sets, associativity, and block size).
- **Instruction Model:** Each core executes a statically defined instruction trace consisting solely of memory accesses, which are either loads (reads) or stores (writes). Each memory access is treated as one instruction. There are no arithmetic or control instructions; only memory references are simulated.
- **Execution Semantics:** A core advances its instruction counter only when the current memory access is successfully completed. If a stall occurs due to a cache miss or bus

unavailability, the instruction is not counted as complete until data is available and the memory operation is resolved.

- **Cache Access Latency:** Cache hits are assumed to complete in one cycle. There is no latency for tag checks or state transitions. Cache misses incur a stall for either memory fetch (100 cycles) or cache-to-cache transfer (2N cycles, where N is the number of words per block), followed by one execution cycle to complete the instruction.
- **Bus Arbitration:** The central bus is modeled as a single shared resource. Only one core can use the bus at a time. If the bus is in use by one core, other cores are blocked from initiating new transactions and must wait. There is no queuing or preemption — a core can only proceed when the bus becomes entirely free.
- **Cache Miss Handling:** On a cache miss, the core first checks whether another cache holds a valid copy of the block. If found in a valid state (M, E, or S), a cache-to-cache transfer is initiated. If not found, data is fetched from memory. Once the data arrives, it is inserted into the cache using an LRU replacement policy.
- **MESI Protocol Enforcement:** The MESI protocol is implemented to maintain coherence between caches. Transitions between Modified, Exclusive, Shared, and Invalid states are handled explicitly via 'set_state()' calls based on the memory operation type and current state of the block across all caches.
- **Write Policy:** The simulator uses a write-back, write-allocate cache policy. On a write miss, the corresponding block is first brought into the cache (allocate), and the write is performed on the cache. Dirty blocks are written back to memory only during eviction. Write hits may trigger invalidations in other caches to preserve coherence.
- **Eviction and Replacement:** When a cache set is full, the least recently used block is selected for replacement. Evictions are only counted when a valid block (in M, E, or S) is replaced. Evicting a block in Invalid state is treated as a no-op and not counted as an eviction.
- **Stall Modeling:** Cores are stalled for the entire duration of a miss (memory or cache transfer), and during bus unavailability. Stalls are tracked per-core using a 'stall[i]' variable. The bus stall time is modeled separately using 'bus_stall'.
- **Idle vs Execution Cycle Accounting:** If a core is stalled (due to cache miss or waiting for bus), it is marked as idle for that cycle. If a core is actively processing a memory instruction (e.g., hit or final resolution of a miss), it is marked as executing.
- **Trace Completion:** Once a core finishes all its instructions, it no longer contributes to simulation cycles. Its idle and execution counters are no longer updated, and it is considered "done."
- **Statistics Collection:** Metrics such as total execution cycles, idle cycles, number of cache misses, evictions, writebacks, invalidations, and bus traffic in bytes are tracked per-core. The maximum execution time across cores is computed for performance analysis.
- **Data Transfer Units:** All data transfers are done at the granularity of cache blocks. Data traffic is accounted in bytes, assuming each block contains 2^b bytes. Transfers include block fetches from memory, writebacks, and inter-cache transfers.
- **Initial State and Tags:** All cache blocks are initialized to the Invalid (I) or Unused (U) state, and tags are set to default values. No prefetching or cache warm-up is simulated.

1.4 Explanation of Output Parameters

- **Total Instructions:** The total number of memory instructions (reads and writes) executed by a core. It corresponds to the length of the trace file for that core.
- **Total Reads / Total Writes:** These count how many read (load) or write (store) operations were performed by each core. The values are extracted by scanning the instruction trace and classifying each operation.
- **Total Execution Cycles:** The number of cycles spent executing its own instruction traces.
- **Idle Cycles:** The number of cycles spent waiting for the resources while other cores are executing its instructions.
- **Cache Misses:** The number of instructions that could not be served from the cache. A miss occurs only once per access (i.e., retries due to bus contention are not counted as new misses). Both read and write misses are included.
- **Cache Miss Rate:** The ratio of cache misses to total instructions, expressed as a percentage. It quantifies the effectiveness of the cache at serving memory accesses without going to memory or other caches.
- **Cache Evictions:** The number of valid blocks removed from the cache to make space for new data. A block in state M, E, or S is considered "evicted." Evicting an invalid block does not count.
- **Writebacks:** The number of times a dirty (Modified) block was written back to memory — either due to eviction or due to a BusRdX. Each writeback is counted once per eviction of a dirty block.
- **Bus Invalidations:** The number of times a core triggered invalidation signals on the bus — due to a write miss when another core has that block and a write hit on a shared block. This metric is tracked per core.
- **Data Traffic (Bytes):** The total amount of data moved by each core over the bus. This includes:
  - Data fetched from memory on a miss.
  - Dirty data written back to memory on eviction.
  - Cache blocks transferred to or from other cores.

  All data traffic is reported in bytes, using block size as the granularity.
- **Total Bus Transactions:** The total number of bus operations initiated across all cores, including read and write misses, writebacks, and cache-to-cache transfers. Each bus action is counted once when initiated.
- **Maximum Execution Time:** The highest value among all four cores' execution cycles. This metric is used to assess how cache parameters affect the critical path of parallel execution, and is plotted in the experimental section.

2 Effect of Cache Parameters on Execution Time

We varied:

- Cache size: 0.5KB,1KB (default), 2KB, 4KB
- Associativity: 1-way, 2-way (default), 4-way, 8-way
- Block size: 16B, 32B (default), 64B, 128B

For each configuration, we measured the **maximum execution time across cores**.

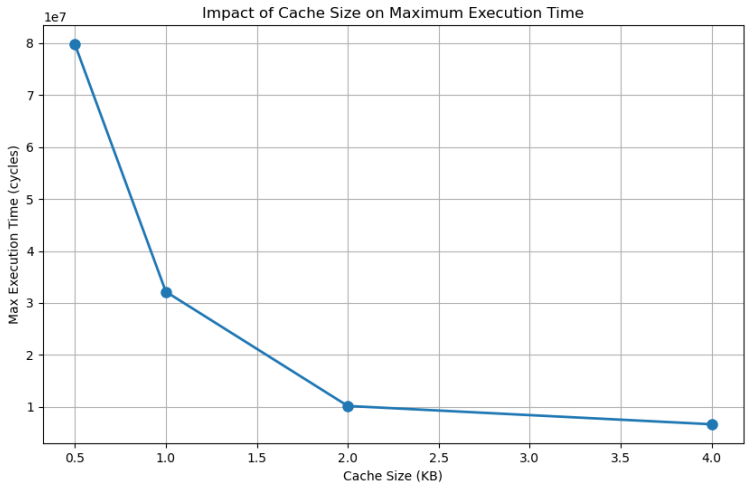

Figure 2: Max execution time vs Cache Size

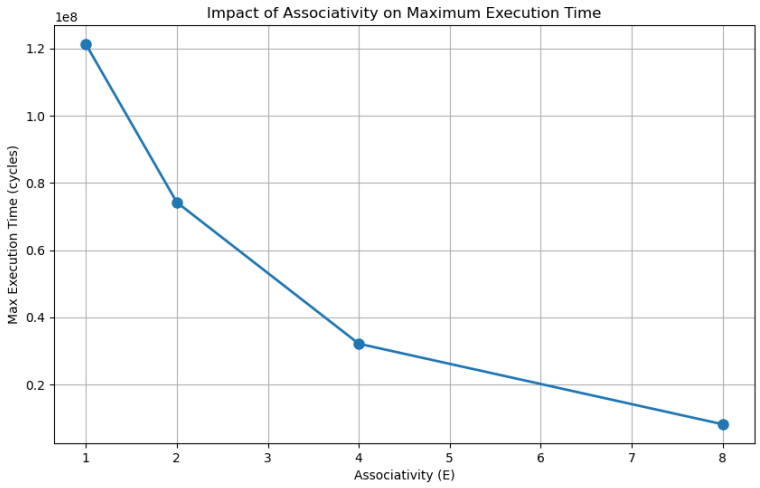

Figure 3: Max execution time vs Associativity

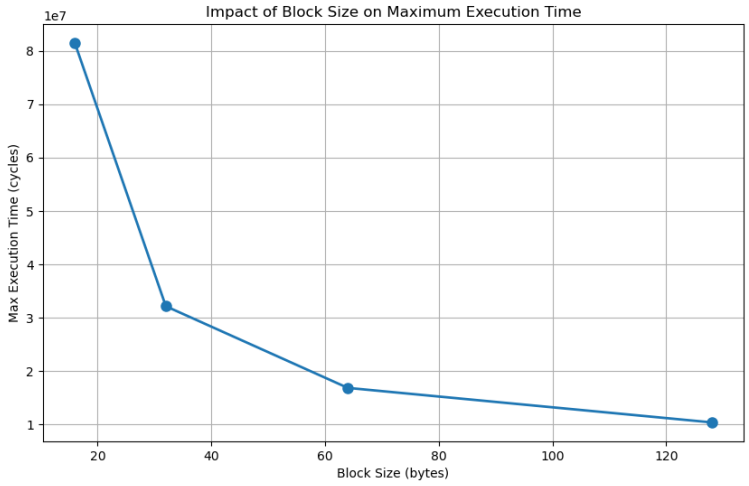

Figure 4: Max execution time vs Block Size

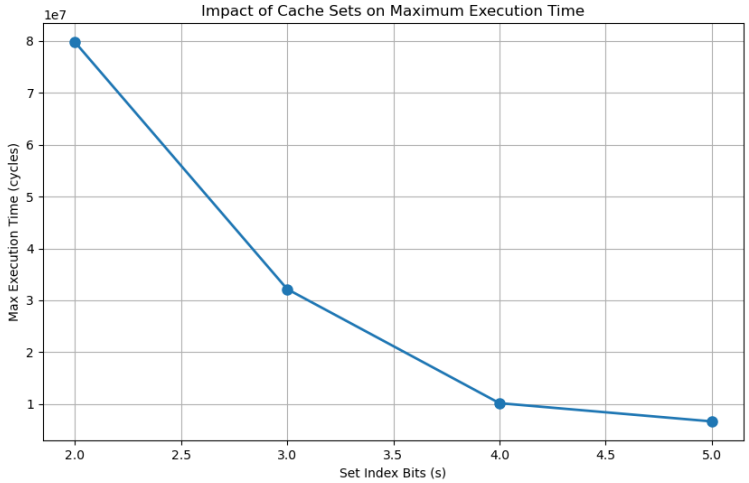

Figure 5: Max execution time vs Cache Sets

2.1 Analysis

- **Increasing cache size significantly reduces execution time.** This is because larger caches reduce the number of cache misses, resulting in fewer long-latency memory fetches. As shown in the cache size graph, execution time drops sharply when cache size increases from 0.5KB to 2KB, but the gain tapers off as the cache grows beyond 2KB.
- **Higher associativity improves performance, especially from 1-way to 4-way.** As associativity increases, conflict misses reduce because multiple blocks can map to the same set. The associativity graph shows a steep reduction in execution time from 1-way to 2-way and again to 4-way, with relatively marginal gains from 4-way to 8-way, indicating diminishing returns.
- **Block size also influences execution time by affecting spatial locality and data traffic.** With a small block size (e.g., 16B), the cache underutilizes spatial locality, causing more frequent misses. As block size increases, execution time improves due to better

prefetching, but the benefit plateaus after 64B–128B as larger blocks increase bus traffic and potential interference. But, an increase in block size can also increase instances of false sharing which can reduce performance.

- **Increasing the number of cache sets (via set index bits) improves execution time by reducing index conflicts.** The execution time drops as more sets are introduced, allowing more distinct addresses to coexist in the cache. However, like other parameters, the benefit diminishes as the set count increases beyond a certain point (seen beyond $s = 4$).
- **All three parameters—cache size, associativity, and block size—demonstrate diminishing returns.** The curves are steep initially and then flatten, which is expected in cache performance scaling.

3 Generated trace files

3.1 False Sharing

We created trace files to demonstrate the case of false sharing. The following experiments have been conducted using the same traces

Experiment 1.1: Parameters — `s = 6`, `E = 2`, `b = 5` (Block size = 32 bytes)

- **Single-Core Run:** When all instructions were executed by Core 0 alone, the simulator reported:

  Maximum execution cycles: `5096`

- **Multi-Core Run with Shared Accesses:** When instructions were distributed across all four cores with 8-byte address offsets between them:

  Maximum execution cycles: `34296`

- **Observation:** Despite parallelization, the total execution time increased. This is due to false sharing — although the cores access different words, those words fall within the same cache block. As a result, coherence traffic (invalidations and write misses) increases dramatically, degrading performance due to excessive inter-core cache line transfers.

Experiment 1.2: Parameters — `s = 6`, `E = 2`, `b = 2` (Block size = 4 bytes)

- **Single-Core Run:** When only Core 0 executed all instructions:

  Maximum execution cycles: `32796`

- **Multi-Core Run with Shared Accesses:** With the same address offset (8 bytes) between concurrent accesses across cores:

  Maximum execution cycles: `32796`

- **Observation:** In this case, the performance remains the same across both runs. Since the block size is smaller than the address offset (4 bytes $<$ 8 bytes), each core accesses data from a different cache block, eliminating false sharing. Hence, there is minimal coherence overhead.

3.2 Multi-threading

We also generated trace files to demonstrate when parallelizing the process actually helps. This happens in case of multiple read or write hits as in the specification for the simulator, at a time only a single transaction can be processed on the bus. This inhibits the performance improvement that is expected from multi-threading the process.

Experiment 2.1: Parameters — `s = 6`, `E = 2`, `b = 5` (Block size = 32 bytes)

- **Single-Core Run:** When all instructions were executed by Core 0 alone, the simulator reported:

  Maximum execution cycles: `24728`

- **Multi-Core Run with Shared Accesses:** When instructions were distributed across all four cores such that independent memory access which would otherwise have overwritten the blocks in the original core are now handled by different processes:

  Maximum execution cycles: `3256`

Conclusion: When memory accesses across threads do not cause coherence conflicts and largely hit in the cache, parallelism can be effectively exploited. The bus remains unused in these cases, and all cores can execute simultaneously without stalling. This demonstrates that under low-conflict workloads, multithreaded execution significantly reduces the overall runtime.

References

- COL216 Assignment 3 Description
- Assignment trace files
- Clarifications on Whatsapp on doc
- Clarifications on Piazza