# L1 Cache Simulator for Quad-Core Processors: Report

**Laksh Goel (2023CS10848)**
**Yuvika Chaudhary (2023CS10353)**

April 30, 2025

## 1 Introduction

We present a C++ simulator for L1 data caches with MESI coherence on a quad-core system. Main goals:

- Simulate 32-bit memory accesses (4-byte word size).
- Model write-back, write-allocate L1 caches with LRU replacement per core, backed by main memory.
- Maintain coherence via a centralized snooping bus implementing MESI.
- Measure per-core performance metrics (cycles, misses, evictions, traffic, etc.).

## 2 Design Decisions

### 2.1 MESI Cache Coherence Protocol

The simulation implements the MESI (Modified, Exclusive, Shared, Invalid) protocol to maintain cache coherence across multiple cores. Each cache line maintains one of these four states, and state transitions are triggered by processor requests and bus snooping operations. The implementation handles both local state changes and remote-induced transitions through bus messages such as `BusRd`, `BusRdX`, and `INVALIDATE`. This approach ensures that multiple caches maintain a consistent view of memory without requiring direct communication between cores.

### 2.2 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.

### 2.3 Bus Requests and Arbitration

The central bus acts as the communication backbone between cores, handling transaction requests in a queue-based FIFO order. The `Bus` class processes one request at a time. Each request undergoes specific timing penalties based on its type (read, write, invalidate) and the current states of all caches. When multiple processors initiate bus transactions simultaneously, arbitration is performed using processor ID — the processor with the lowest ID wins.

### 2.4 LRU Replacement

The cache replacement policy uses a Least Recently Used (LRU) algorithm implemented through `CacheSet`'s linked list structure.
When a new cache line is accessed, it's moved to the front of the list, while evictions target the least recently used line at the back.
The `splice` operation efficiently updates the recency ordering without requiring full list traversals.

### 2.5 Memory Model and Timing

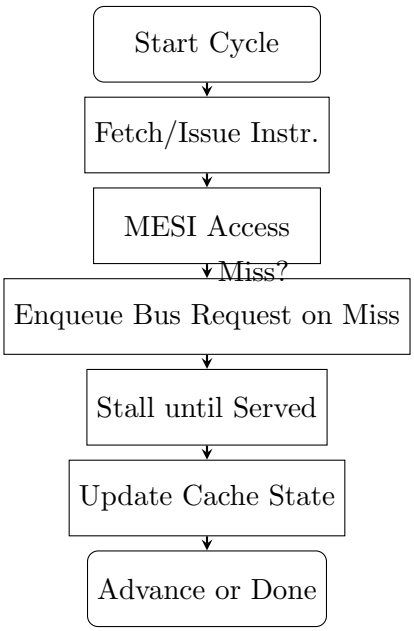
The simulation implements a detailed memory timing model that accounts for different latencies across various operations. The `Request` class tracks timing components including eviction time, memory access time, and cache-to-cache transfer time. The following specific timing parameters are used:

- 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 (in Modified state) costs 100 cycles

### 2.6 Blocking Caches

The implementation uses blocking caches, where a processor stalls (enters a halted state) when it issues a request that results in a cache miss. The processor remains halted until the request completes. However, it continues to service coherence requests (snooping) from the bus. The `haltProcessor` method in the `Bus` class manages this processor stalling mechanism, while the state machine in `Processor` class captures different execution states including `FREE`, `READ_MEMORY`, and `WRITE_MEMORY`, providing realistic modeling of execution delays.

### 2.7 Flow Diagram



# 3 Assumptions

- **Core and Instructions:** The simulation models a quad-core processor system, with each core having a private, exclusive L1 data cache. There is no shared cache. Each memory access is treated as one instruction. There are no arithmetic or control instructions; only memory references are simulated.
- **Processor execution:** A core advances it's instruction counter only when current Instruction is 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.
- **Bus Arbitration:** There is a single Shared bus. All data transfers between Caches and Memory is there through bus. Only one core can use bus at a time(i.e. only one request is processed at a time). We have a queing model, where all requests are queued and are completed as per Cache Access Latency. While a request of a processor is queued, that processor cannot move to next Instruction until it's request is completed.
- **Cache Miss Handling:**
  1. First Bus Checks if there is a valid copy(M, E or S) in other Caches. If found, a Cache to Cache transfer is initiated with a state change of other caches according to Fig: RemotelyInitiatedChanges.
  2. If not found, data is read from memory. Once data arrives, it is inserted into Cache using LRU replacement policy.
  3. Entire process can take at max. 100 (for write back in case of M) + 2*N + 100 (LRU replacement in case of M) cycles, of which, first 100 are counted in IdleCycles, and next 2*N + 100 in execution cycles.
- **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.
- **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.
- **Initial State and Tags:** Initially, Valid variable of Cache lines is set to false (i.e. no data present). And state of each Cacheline is considered Invalid. No Cache-warmup is implemented.
- **Halt Modelling**:
  1. If there is a
     - Cache Miss
     - Write Hit (with data in S state) and Bus is busy

     then the Processor has to Halt to complete it's Request.
  2. If the Processor is halted, we don't make any change in Processor (Cycle of processor not called) other than changing it's Execution Cycle and Idle Cycles.
  3. Processor is un-halted if it's request is Completed and goes to next Instruction.

- **Jump Modelling:** If all the Processsors are Halted, then we consider it as a Jump. We jump the number of cycles by the amount of time left for the top request to be executed. Also, ExecutionCycles and IdleCycles of all processors are also updated.
- **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."
- **Idle Cycle vs Execution Cycle:** Whenever the core is waiting for Resources, it is considered as Idle Cycle. For example
  1. Core want's to send request to bus, but bus processing request of some other Cache.
  2. In my own request, I'm taking data from other Cache, but that data was M. So writing back that data to Memory is counted in Idle Cycles of this Processor

  If the Core is processing it's won request, it is considered in ExecutionCycles. It also includes Evicting it's own Cache due to LRU policy.

# 4 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 counted while scanning trace file in each core.
- **Total Execution Cycles:** The number of cycles spent executing its own instruction traces, also includes LRU replacement cycles. Don't include Write-back os other Caches.
- **Idle Cycles**: The number of cycles spent waiting for the resources while other cores are executing its instructions.
- **Cache Misses**: The total number of memory accesses that were not satisfied by the cache. Each access results in at most one miss, regardless of any retry attempts caused by bus contention. This includes misses for both read and write operations.
- **Cache Miss Rate**: The percentage of total instructions that resulted in a cache miss. This metric reflects how efficiently the cache handles memory accesses without needing to access main memory or other caches.
- **Cache Evictions**: The count of valid cache blocks that were replaced to accommodate new data. Only blocks in the M (Modified), E (Exclusive), or S (Shared) states are considered evicted; removing invalid blocks does not count. This also includes evictions due to a full cache.
- **WriteBacks:** Number of times a Modified block is written back to memory.
- **Bus Invalidations:** This is the number of times a processor send an INVALIDATE signal or an RWITM signal. In both these case, if data is present in other Caches with same tag and Index, then that CacheLine would be set to Invalidate state.
- **Data Traffic:** The total amount of data moved by each core over the bus. This includes:
  1. Data fetched from Memory on Miss

2. Cache blocks transferred to or from other cores.
3. Modified blocks written back to Memory.

- **Total Bus Transactions**: The cumulative number of bus operations triggered by all cores. This includes read and write misses, writebacks, and cache-to-cache transfers. Each transaction is counted once at the time it is initiated.
- **Total Bus Traffic**: The total volume of data transmitted over the bus during the simulation.

# 5 Custom Trace Files

We made trace files showing Simulation of our Program, including trace files for false sharing.

- **our1, our2 and our3 traces**: These are simple test cases to demonstrate correct cache coherence of our simulator. We are checking for different cases of read hits, read misses, write hits and write misses and the small size of our1 and our2 traces ensure that we can correctly check our simulator.
- **false1 and false2 traces**: To demonstrate false sharing, we have added two test cases namely `false1` and `false2`. These test cases consist of memory access patterns from multiple cores that operate on distinct variables residing within the same cache block when the block size is large (e.g., 32 bytes for `b = 5`). Due to the MESI protocol, even though the cores are not sharing data semantically, they cause unnecessary invalidations and transitions in cache states, leading to performance degradation — a classic symptom of false sharing.

  To contrast this behavior, we also include corrected versions `false1c` and `false2c` in our Makefile, which use a smaller block size (8 bytes for `b = 3`) so that these distinct variables map to different cache blocks. This eliminates false sharing, resulting in significantly fewer cache invalidations and writebacks, and hence lower data traffic.

  The simulation outputs clearly show a higher number of invalidations and increased data traffic for `false1` and `false2` compared to their corrected versions, thereby empirically validating the presence of false sharing in the original cases.

# 6 Parameter Sensitivity Study

We varied Associativity, Cache Size and Block Size

1. Cache Size: 0.25KB, 0.5KB (default), 1KB , 2KB
2. Associativity: 1, 2 (default), 4, 8
3. Block Size: 16B, 32B (default), 64B, 128B

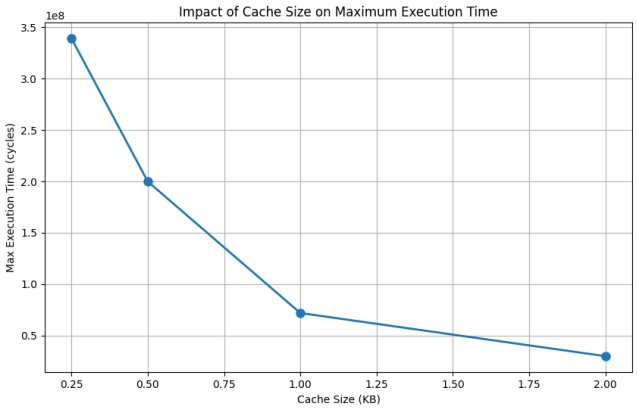



Figure 1: Varying Cache Size for app1

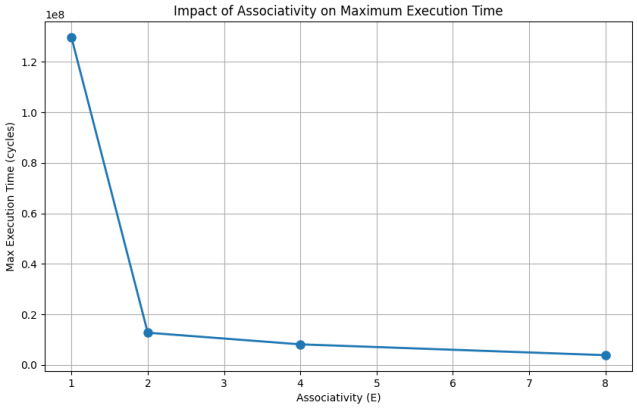



Figure 2: Varying Associativity for app1

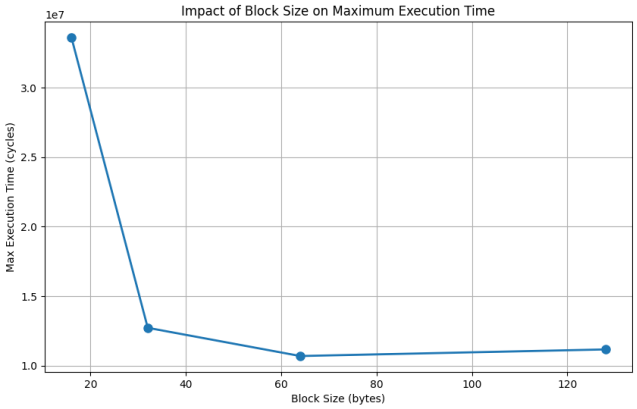



Figure 3: Varying Block Size for app1

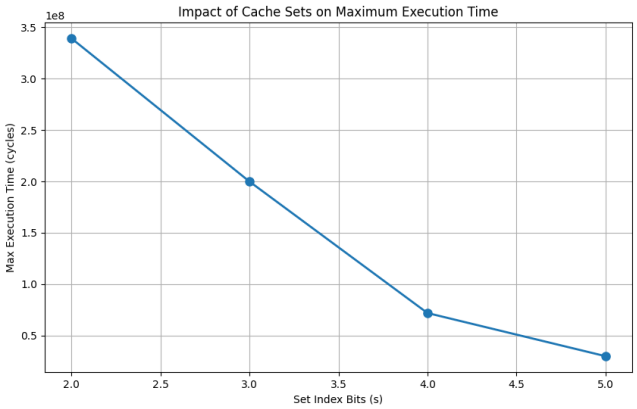



Figure 4: Varying Cache sets for app1

# 7 Conclusion

- **Cache Size**: 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. From the graph, increasing cache size singificantly reduces maximum execution time.
- **Block size**: Block size affects execution time by affecting spatial locality. With larger block size, there is more spatial locality, hence hits will be more. Although increasing block size can also lead to increase in false sharing cases.
- **Cache sets**: Increasing the number of cache sets improves maximum execution time. More are the number of sets, more distinct addresses can co-exist. And also, more data can be stored in the cache.
- **Associativity**: Higher associativity improves performance. Multiple blocks can map to same set, so conflict misses reduce. Graph shows a steep reduction in execution cycles as we go from 1 to 2 in associativity showing great increase in speed.