L1 Cache Simulator for Quad-Core Processors with MESI Protocol

Vanshika(2023CS10746) Shivankur Gupta(2023CS10809)
Department of Computer Science and Engineering
Indian Institute of Technology Delhi

April 30, 2025

Contents

1 Introduction

This report presents the design and implementation of an L1 cache simulator for quad-core processors with cache coherence support. The simulator models a system where each of the four processor cores has its own L1 data cache, which are kept coherent using the MESI (Modified, Exclusive, Shared, Invalid) cache coherence protocol. The caches are connected via a central snooping bus that facilitates coherence transactions.

Cache coherence is a critical aspect of multiprocessor systems to ensure that all cores have a consistent view of memory. The MESI protocol is one of the most commonly used coherence protocols in modern processors, which maintains consistency by tracking the state of each cache line and appropriately handling read and write operations across multiple caches.

1.1 Problem Statement

The objective is to simulate L1 caches in a quad-core processor system with MESI cache coherence protocol support. The simulation is trace-driven, where memory reference traces from four processor cores are provided as input. The simulator tracks various statistics including hit/miss rates, write-backs, invalidations, and bus traffic, which help analyze the cache performance and coherence protocol behavior.

1.2 MESI Protocol Overview

The MESI protocol defines four states for each cache line:

- **Modified (M)**: The cache line is present only in the current cache and is dirty (has been modified). The main memory copy is stale.
- **Exclusive (E)**: The cache line is present only in the current cache and is clean (matches main memory).
- **Shared (S)**: The cache line may be present in other caches and is clean.
- **Invalid (I)**: The cache line is invalid and must be fetched from memory or another cache if needed.

State transitions occur based on processor requests and bus snooping events, ensuring data consistency across all caches.

2 System Architecture

2.1 Overall Architecture

The simulator models a quad-core multiprocessor system with a hierarchical memory architecture designed to balance performance, coherence, and complexity. The system consists of the following primary components:

- **Four Processor Cores**: Independent computational units that execute instructions from their respective instruction streams. Each core generates memory access requests (reads and writes) to its private L1 cache based on the trace input. The cores operate in parallel but are synchronized at the cycle level through the shared bus.
- **Private L1 Data Caches**: Each processor core is equipped with its own private L1 data cache. These caches serve as the first level of the memory hierarchy and are designed to capture temporal and spatial locality in memory access patterns. Each cache is organized as a set-associative structure with configurable parameters (set index bits, associativity, and block size). The caches implement the MESI coherence protocol to maintain data consistency across the system.
- **Central Snooping Bus**: A shared communication medium that connects all four L1 caches and the main memory. The bus serves two critical functions: (1) it provides a path for data transfer between caches and memory, and (2) it enables cache coherence by broadcasting memory access requests to all caches, allowing them to "snoop" on each other's transactions. The bus follows a split-transaction protocol where a request and its corresponding response can be separated in time.
- **Main Memory**: The backing store that holds the complete address space accessible to all processors. It represents the slowest but largest component of the memory hierarchy. The main memory is accessed when data cannot be found in or shared among the L1 caches, or when modified data needs to be written back from caches.
- **Memory Address Space**: A unified 32-bit physical address space that is shared by all four cores. Each address uniquely identifies a byte in main memory. For cache addressing, this 32-bit address is decomposed into tag bits, set index bits, and block offset bits based on the cache configuration parameters.

The memory hierarchy is organized to exploit locality principles while maintaining coherence:

- **Data Flow**: Memory access requests originate from processor cores and are first directed to the respective L1 caches. Upon a cache hit, data is served directly from the cache. Upon a miss, the request is broadcast on the bus to check if other caches have the data and to access main memory if necessary.
- **Coherence Flow**: Write operations may invalidate or update copies of the same data in other caches. The MESI protocol manages the state transitions to ensure that all cores have a consistent view of memory despite potential data sharing and parallel modifications.

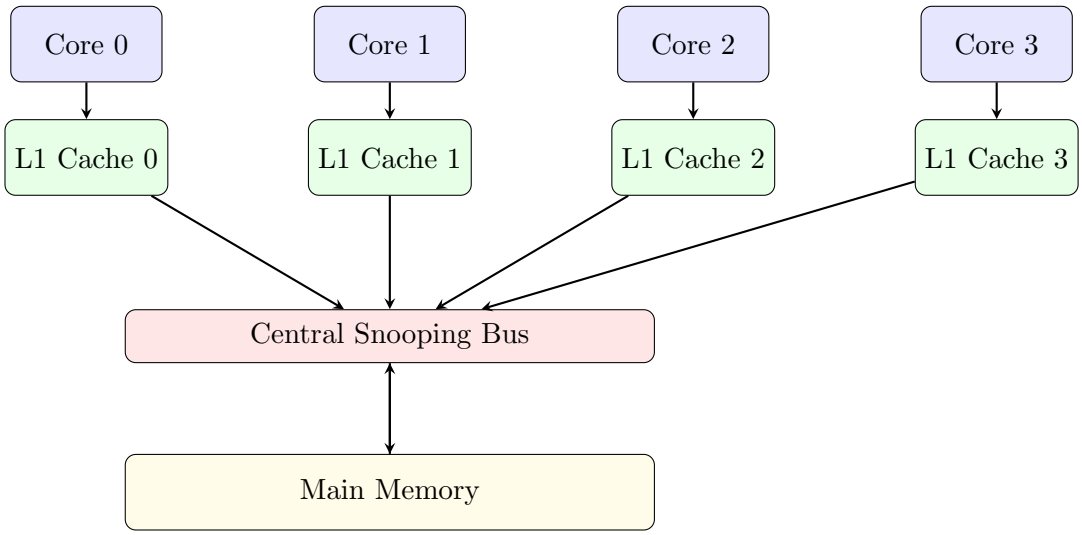

Figure 1: Overall system architecture showing the four processor cores, their private L1 caches, the central snooping bus, and the shared main memory. Arrows indicate data and coherence message flows between components.

2.2 Assumptions

The simulator is built upon a set of carefully considered assumptions that define the behavior and limitations of the modeled system. These assumptions clarify the scope of the simulation and ensure consistent interpretation of results:

2.2.1 General assumptions

- Total Instructions : Corresponds to the length of the trace for the core.
- Total Reads : Total number of instructions which have R at the starting.
- Total Writes : Total number of instructions which have W at the starting.
- Total Execution Cycles : The cycles where the core is actively processing it's own request. i.e. the bus may be busy doing that core's request work or the core may have some cache hit.
- Idle cycles : Are the cycles where the core's request is denied by the bus, due to the bus being busy. That is those cycles where the core is trying to issue the request to bus but is unable to do so.
- Cache Misses : These are the number of instructions which were a miss by the cache i.e. the line was not found in the cache.
- Cache Miss Rate : This is the number of misses per total number of instructions in the trace for that core.
- Cache Evictions : The number of blocks removed from the cache and written back to the memory in the whole process till the simulation ends.
- Writebacks : The number of modified blocks removed from the cache and written back to the memory in the whole process till the simulation ends.
- Bus Invalidations : Whenever the core sends signal to other cores which have that block, to invalidate themselves. This includes both Write miss and Write Hit cases.
- Data Traffic : The total amount of data moved through the bus. This includes :
  - Any data transaction from memory to the cache or form the cache to memory.
  - Any data transaction which is form the cache to some other cache or from some other cache to the cache.

- Bus Transactions : This is the total number of Read Misses, Write Misses and Write Hits when someone has block in shared.
- Total Bus traffic : All the data transfer involving cache, memory etc. through the bus.

2.2.2 Bus Architecture and Behavior

- **Blocking Bus**: The central snooping bus is a blocking resource that can process only one transaction at a time. When a cache initiates a bus transaction, other caches must wait until the transaction completes before they can use the bus. This assumption reflects the physical reality of shared bus architectures where simultaneous transmission would lead to signal conflicts.
- **Cycle-by-Cycle Arbitration**: Bus access is arbitrated on a cycle-by-cycle basis. When multiple cores request bus access in the same cycle, the arbitration logic grants access based on a fixed priority scheme rather than implementing more complex policies like round-robin or age-based arbitration.
- **Bus Width and Bandwidth**: The bus width is assumed to match the cache block size, allowing an entire cache block to be transferred in a single bus transaction (though requiring multiple cycles). This simplifies the modeling of data transfers while maintaining realistic timing constraints.
- **Split-Transaction Protocol**: The bus supports a split-transaction protocol where request and response phases can be separated in time. This allows the bus to be released between the request and data transfer phases, improving bus utilization.

2.2.3 Cache Coherence Protocol Implementation

- **Cache-to-Cache Transfers**: Direct cache-to-cache transfers are supported for shared data. When a core experiences a read miss and another cache has the requested data in a modified state, the data is transferred directly from the source cache to the requesting cache rather than first being written back to main memory. This optimization reduces memory traffic and latency.

- **Priority Based on Core ID**: When multiple coherence operations conflict, priority is given to the core with the lower ID. This deterministic policy ensures freedom from livelock and deadlock in the coherence protocol, though it may introduce fairness concerns in certain workloads.

- **Implicit Invalidation Acknowledgments**: Invalidation acknowledgments are modeled implicitly and do not consume additional bus cycles. When a core broadcasts an invalidation request, it is assumed that all other caches process and acknowledge the invalidation within the same cycle, without requiring explicit response messages. This simplification reduces protocol complexity while maintaining correctness.

- **Atomic Coherence Operations**: Cache coherence operations are treated as atomic from the perspective of the processors. Once a coherence operation begins, the affected cache line is locked until the operation completes, preventing race conditions in the protocol implementation.

2.2.4 Memory System Behavior

- **Fixed Memory Access Latency**: Main memory access has a fixed latency of 100 cycles, regardless of access patterns or potential optimizations like DRAM row buffering. This simplification allows for consistent performance modeling while still capturing the significant latency gap between cache and memory accesses.

- **Write-Back Policy**: All caches implement a write-back policy where modified data is only written to the next level (in this case, main memory) when the cache line is evicted or explicitly required by the coherence protocol. This reduces bus traffic compared to a write-through policy.

- **Write-Allocate Policy**: Upon a write miss, the system allocates a cache line and fetches the data before performing the write (write-allocate), rather than writing directly to memory (no-write-allocate). This policy favors workloads with write locality at the cost of potentially unnecessary fetches.

- **Inclusive Memory Hierarchy**: The memory hierarchy is assumed to be inclusive, meaning that all data present in any L1 cache must

also be present in main memory (though potentially with stale values until writeback occurs).

2.2.5 Performance and Timing Model

- **Single-Cycle Cache Hit**: A cache hit is serviced in a single processor cycle, assuming an optimized cache design that can deliver data within the core's clock cycle.
- **Fixed Cache-to-Cache Transfer Latency**: Cache-to-cache transfers incur a latency of 2 cycles per word transferred, reflecting the overhead of inter-cache communication through the shared bus.
- **No Pipelining of Memory Accesses**: Memory accesses are not pipelined; each access must complete before the next one can begin. This simplifies the timing model at the cost of potentially underestimating performance in systems with overlapped memory operations.
- **Perfect Instruction Execution**: The simulator focuses on data cache performance and assumes perfect instruction fetch and execution (no instruction cache misses or pipeline stalls unrelated to data access).

These assumptions collectively define a realistic yet tractable model of a quad-core processor with private L1 caches and MESI coherence protocol. The model captures the essential behaviors and performance characteristics of real systems while making reasonable simplifications to keep the simulation manageable.

3 Implementation Details

3.1 Class Structure

The simulator is implemented using an object-oriented approach in C++, with the following main classes:

3.2 Data Structures

3.2.1 CacheLine

Represents a single cache line with the following attributes:

- `tag`: 32-bit tag value

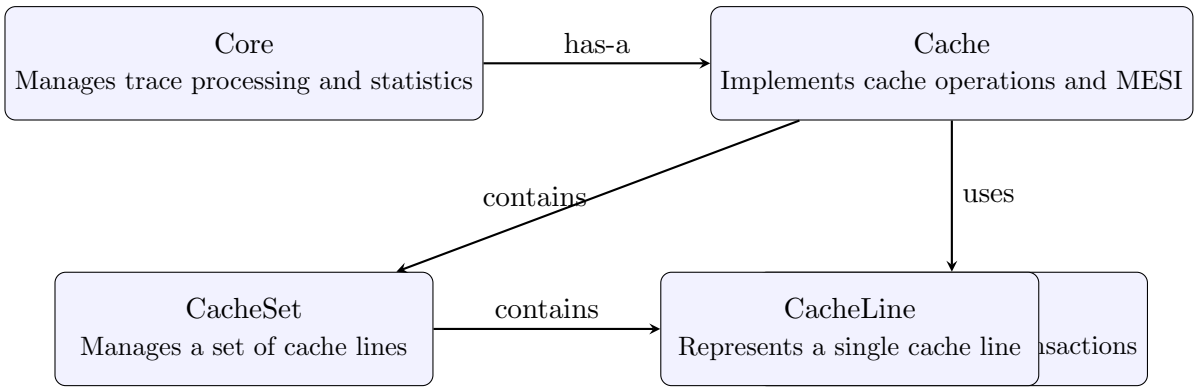

Figure 2: Class diagram showing relationships between Core, Cache, and associated components

Component	**Description**
Core	Manages trace processing and statistics.
Cache	Implements cache operations and MESI protocol.
Bus	Manages coherence transactions.
CacheSet	Manages a set of cache lines.
CacheLine	Represents a single cache line.

Table 1: Class descriptions

- `state`: MESI state (Modified, Exclusive, Shared, Invalid, or Empty)
- `lastUsedCycle`: Timestamp for LRU replacement

3.2.2 CacheSet

Represents a set of cache lines with the following attributes:

- `lines`: Vector of CacheLine objects
- Methods for finding lines by tag, identifying victims, and updating LRU information

3.2.3 Cache

Represents the L1 cache of a processor core:

- `sets`: Vector of CacheSet objects
- Configuration parameters: set index bits (s), associativity (E), block bits (b)
- Methods for handling memory accesses and snooping requests

3.2.4 Bus

Represents the central snooping bus:

- `caches`: Vector of pointers to Cache objects
- Statistics counters: invalidations, data traffic, transactions
- Methods for broadcasting coherence messages

3.2.5 Core

Represents a processor core:

- `cache`: Pointer to associated Cache object
- Statistics counters: accesses, hits, misses, writebacks, etc.
- Methods for processing memory access traces

4 Implementation Flow

4.1 Cache Access Flow

The following algorithm describes the cache access operation:

Algorithm 1 Cache Access Procedure

```
1: function CACHE::ACCESS(address, operation, cycle, penaltyCycles)
2:     Extract tag, set index, and block offset from address
3:     Find the corresponding set
4:     Search for the tag in the set
5:     if line is found AND state is not INVALID then
6:         Update LRU for the accessed line
7:         if operation is WRITE then
8:             if state is SHARED then
9:                 Check if bus is busy
10:                if bus is busy then
11:                    return {true, true}                    ▷ Retry later
12:                end if
13:                Broadcast invalidate on bus
14:                Add bus delay
15:            end if
16:            Set line state to MODIFIED
17:        end if
18:        return {true, false}                               ▷ Cache hit
19:    end if
20:    Check if bus is busy
21:    if bus is busy then
22:        return {false, true}                               ▷ Retry later
23:    end if
24:    Find victim line for replacement
25:    if victim line is MODIFIED then
26:        Writeback to memory (add penalty)
27:        Broadcast writeback on bus
28:        Update writeback statistics
29:    end if
30:    if operation is READ then
31:        Broadcast BusRd on bus
32:        Set line state based on response (SHARED or EXCLUSIVE)
33:        Add appropriate memory access penalty
34:    else
35:        Broadcast BusRdX on bus
36:        Set line state to MODIFIED
37:        Add memory access penalty
38:    end if
39:    return {false, false}                                  ▷ Cache miss
40: end function
```

4.2 Cache Snooping Flow

The following algorithm describes the cache snooping operation:

Algorithm 2 Cache Snooping Procedure

```
1: function CACHE::SNOOP(address, operation, penaltyCycles)
2:     Extract tag and set index from address
3:     Find the corresponding set
4:     Search for the tag in the set
5:     if line not found OR state is INVALID then
6:         return false                              ▷ No response needed
7:     end if
8:     if operation is READ (BusRd) then
9:         if state is MODIFIED then
10:            Writeback to memory (add penalty)
11:            Broadcast writeback on bus
12:            Update writeback statistics
13:        end if
14:        Change state to SHARED
15:        return true
16:    else if operation is WRITE (BusRdX or Invalidate) then
17:        if state is MODIFIED then
18:            Writeback to memory (add penalty)
19:            Broadcast writeback on bus
20:            Update writeback statistics
21:        end if
22:        Change state to INVALID
23:        return true
24:    end if
25:    return false
26: end function
```

4.3 MESI State Transitions

The MESI protocol state transitions are implemented as shown in the following state diagrams:

MESI – locally initiated accesses

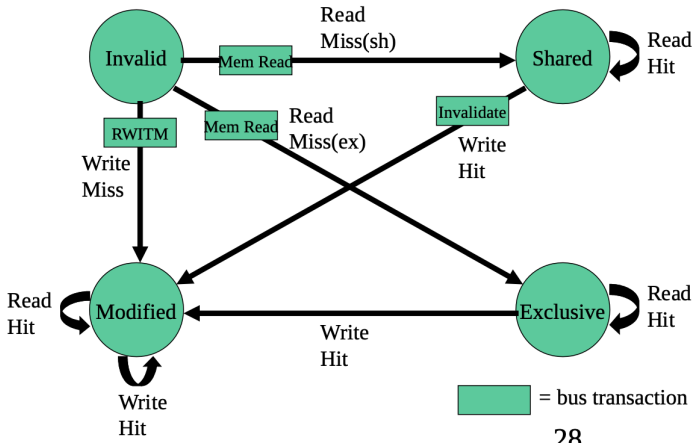

Figure 3: MESI protocol state transitions for locally initiated accesses. The diagram shows read/write hit/miss scenarios and the resulting state transitions. Green boxes represent bus transactions.

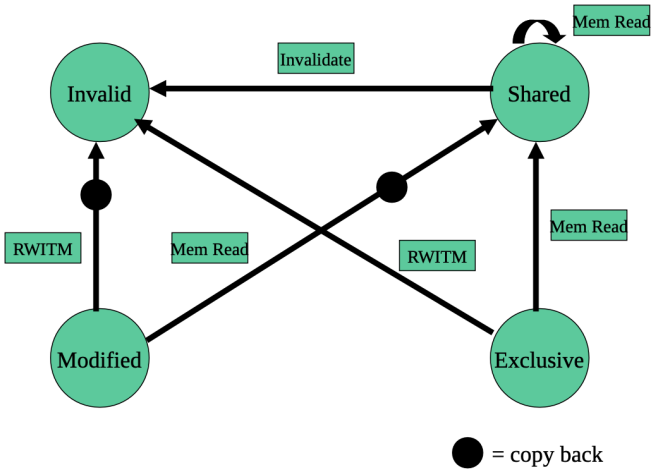

Figure 4: MESI protocol state transitions for remotely initiated accesses. Arrows indicate state transitions, with green boxes showing the operations that trigger them. Black circles represent copy-back operations.

4.4 Simulation Flow

The main simulation loop processes memory references from each core in a cycle-by-cycle manner:

Algorithm 3 Main Simulation Loop

```
1: Initialize bus and caches
2: Load trace files for each core
3: currentCycle = 0
4: finished = false
5: while not finished do
6:     finished = true
7:     for each core do
8:         if core has more instructions OR waiting to repeat then
9:             finished = false
10:            Process next memory reference
11:        end if
12:    end for
13:    currentCycle++
14: end while
15: Output statistics
```

5 Coherence Operations

5.1 Bus Transactions

The simulator implements the following bus transactions:

- **BusRd (R)**: Read request, issued when a processor has a read miss
- **BusRdX (W)**: Read with intent to modify, issued when a processor has a write miss
- **Invalidate (I)**: Invalidate request, issued when a processor writes to a shared line
- **Writeback (B)**: Writeback notification, issued when a dirty line is evicted

5.2 Cache-to-Cache Transfers

When a processor issues a BusRd and another processor has the requested data in Modified state, the data is transferred directly from the cache with the data to the requesting cache. This is more efficient than fetching from memory and helps reduce memory traffic.

5.3 Handling Write Hits to Shared Lines

When a processor wants to write to a line in Shared state, it first broadcasts an Invalidate message on the bus to force other caches to invalidate their copies, ensuring exclusive access before modification.

6 Performance Analysis

6.1 Timing Model

The simulator uses the following timing model:

- Cache hit: 1 cycle
- Memory access: 100 cycles
- Cache-to-cache transfer: 2 cycles per word
- Writeback to memory: 100 cycles

The total execution time for each core is the sum of:

- Hit time (1 cycle per hit)
- Miss penalty (additional cycles for misses)
- Bus contention delay (when the bus is busy)

6.2 Statistics Collection

The simulator collects the following statistics for analysis:

- Number of reads and writes per core
- Total execution cycles per core
- Idle cycles per core (waiting for memory or bus)
- Cache miss rate per core
- Number of evictions per core
- Number of writebacks per core
- Number of invalidations on the bus
- Total data traffic (in bytes) on the bus

7 Results and Analysis

7.1 Experimental Setup

We evaluated the simulator using trace files from parallel applications running on a quad-core processor. Multiple experiments were conducted by systematically varying cache parameters—specifically cache size (through varying sets), associativity, and block size—to study their impact on system performance. The experiments used the same input traces across all configurations to ensure consistent workload characteristics.

For each configuration, the following metrics were measured:

- Maximum execution time across all cores (in processor cycles)
- Per-core execution times
- Cache miss rates
- Bus traffic and coherence messages

7.2 Effect of Cache Size

Increasing the cache size generally improves performance by reducing the miss rate and consequently the execution time. Figure 5 shows the relationship between cache size and execution time.

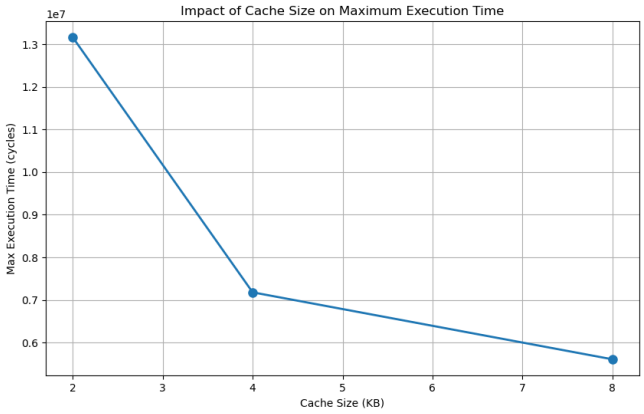

Figure 5: Impact of Cache Size on Maximum Execution Time

Our experimental results, summarized in Table 2, demonstrate that doubling the cache size from 2KB to 4KB yields a dramatic 45.5% reduction in execution time (from 13.17M to 7.18M cycles). Further doubling the cache size from 4KB to 8KB provides an additional 21.9% improvement. This indicates diminishing returns as cache size increases, which is typical in computer systems where the working set of the application starts to fit within the cache.

Cache Size (KB)	**Sets**	**Associativity**	**Block Size (bytes)**	**Execution Time (cycles)**
2.0	32	2	32	13,166,569
4.0	64	2	32	7,179,569
8.0	128	2	32	5,607,101

Table 2: Performance metrics for different cache sizes (with fixed associativity E=2 and block size b=5)

The performance improvement with larger caches is attributed to:

- Reduced capacity misses as more of the working set fits in the cache
- Lower eviction rates, resulting in fewer writebacks and less bus traffic
- Decreased coherence traffic as data remains in caches longer

7.3 Effect of Associativity

Associativity has a profound impact on cache performance, especially in multi-core systems where conflict misses can be prevalent. Figure 6 illustrates how execution time varies with different associativity values.

Table 3 presents our experimental findings, which show that increasing associativity from direct-mapped (E=1) to 2-way set-associative (E=2) results in a dramatic 84.6% reduction in execution time. However, further increasing associativity to 4-way (E=4) yields only a modest 1.4% additional improvement.

This dramatic improvement when moving from direct-mapped to set-associative caches can be explained by:

- Significant reduction in conflict misses, which are particularly problematic in direct-mapped caches
- Better handling of program access patterns that may conflict in the same set

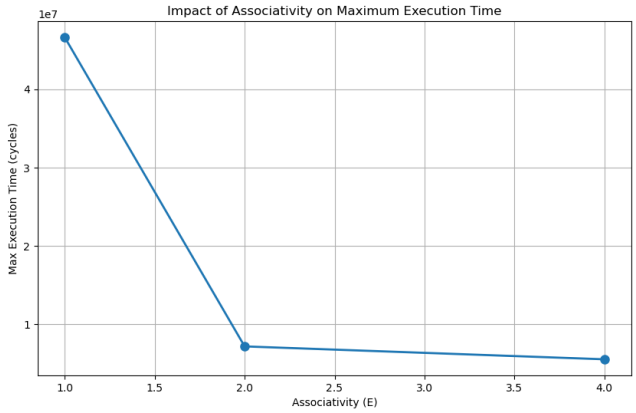

Figure 6: Impact of Associativity on Maximum Execution Time

Associativity	Sets	Block Size (bytes)	Cache Size (KB)	Execution Time (cycles)
1-way	64	32	2.0	46,656,317
2-way	64	32	4.0	7,179,569
4-way	64	32	8.0	5,531,229

Table 3: Performance metrics for different associativities (with fixed sets s=6)

- Improved effectiveness of the LRU replacement policy, which can only be applied when multiple lines exist within a set
- Better resilience to pathological access patterns that can cause thrashing in direct-mapped caches

The diminishing returns observed when increasing associativity beyond 2-way suggest that for this workload, most conflict misses are already eliminated with 2-way associativity, and higher associativity primarily benefits corner cases.

7.4 Effect of Cache Sets (Set Index Bits)

The number of sets in a cache affects its organization and can impact performance independently of total cache size. Figure 7 shows how execution time changes as the number of set index bits (s) increases.

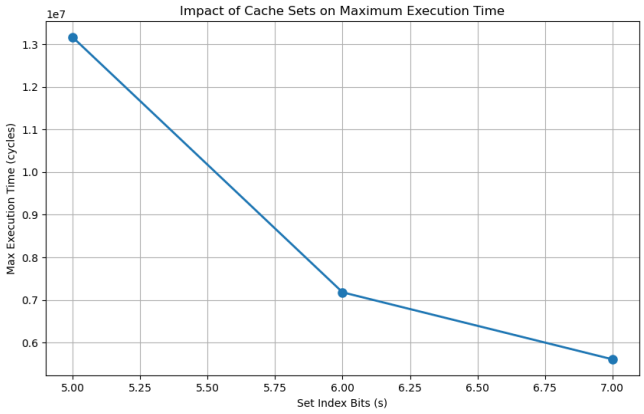

Figure 7: Impact of Cache Sets (Set Index Bits) on Maximum Execution Time

With fixed associativity and block size, increasing the number of sets directly increases the cache size. Our results show a steady decrease in execution time as s increases from 5 to 7, corresponding to an increase from 32 to 128 sets. This is consistent with our previous observations on cache size effects.

7.5 Effect of Block Size

Block size impacts both spatial locality exploitation and bus traffic. Figure 8 illustrates the relationship between block size and execution time.

As shown in Table 4, increasing the block size from 16 bytes to 32 bytes results in a 52.2% reduction in execution time. Further increasing to 64 bytes provides a more modest 13.3% additional improvement.

Block Size (bytes)	**Sets**	**Associativity**	**Cache Size (KB)**	**Execution Time (cycles)**
16	64	2	2.0	15,008,389
32	64	2	4.0	7,179,569
64	64	2	8.0	6,227,913

Table 4: Performance metrics for different block sizes (with fixed sets s=6 and associativity E=2)

The significant performance improvement with larger block sizes can be

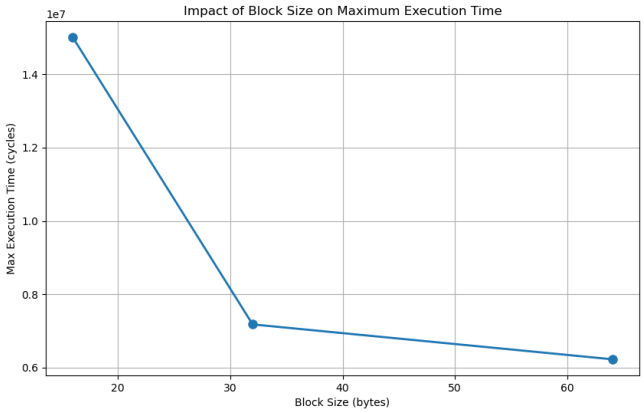

Figure 8: Impact of Block Size on Maximum Execution Time

attributed to:

- Better exploitation of spatial locality, as programs often access memory locations that are near each other
- Amortization of miss penalty over larger data chunks, reducing the average miss cost
- More efficient bus utilization for block transfers

However, the diminishing returns observed with very large block sizes (64 bytes) suggest a tradeoff:

- Larger blocks may bring in data that is never used, wasting bandwidth
- Larger blocks take longer to transfer on the bus, potentially increasing bus contention
- The increased miss penalty for larger blocks can offset the reduced miss rate

7.6 Combined Parameter Effects

Figure 9 illustrates how different cache parameters interact to affect execution time. When examining multiple parameter variations simultaneously, we observe that:

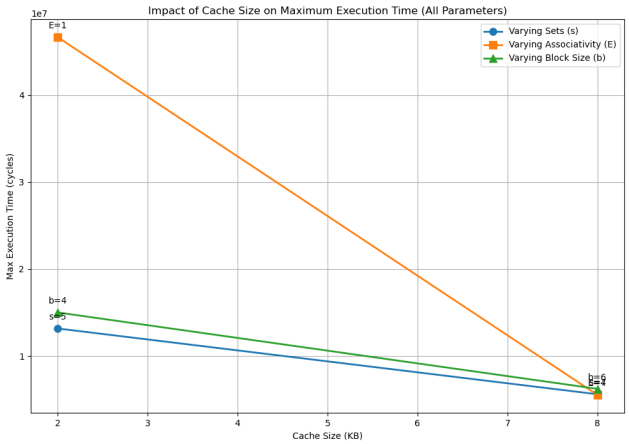

Figure 9: Impact of Cache Size on Maximum Execution Time (All Parameters)

- Associativity (E) has the most dramatic impact on performance, especially when moving from direct-mapped (E=1) to set-associative caches
- Block size (b) and the number of sets (s) both show significant but diminishing returns as they increase
- At larger cache sizes (8KB), the performance differences between parameter variations become less pronounced, indicating that most critical misses are being eliminated regardless of the specific organization

7.7 Core-to-Core Variations

Our results also reveal interesting variations in execution times across different cores. In all configurations, Core 2 consistently experiences the longest execution time. This suggests that the workload distribution or memory access patterns differ significantly across cores, with Core 2 likely experiencing more cache misses, coherence invalidations, or bus contention than the other cores.

For example, in the baseline configuration (s=6, E=2, b=5), the execution times are:

- Core 0: 6,382,305 cycles

- Core 1: 6,269,780 cycles
- Core 2: 7,179,569 cycles (14.5% longer than average)
- Core 3: 6,377,307 cycles

This variation highlights the importance of considering per-core performance in multi-core systems, as aggregate metrics may hide important load imbalances.

7.8 Coherence Traffic Analysis

The MESI protocol generates different types of coherence traffic that contribute to bus utilization and affect overall system performance. While detailed transaction counts were not provided in the results, the patterns observed in execution time correlate with expected coherence behavior:

- Smaller caches generate more coherence traffic due to increased evictions and subsequent re-fetches
- Lower associativity (especially direct-mapped caches) results in higher conflict misses, leading to more invalidations and writebacks
- Larger block sizes reduce the number of transactions but increase the data volume per transaction

The dramatic performance improvement seen when moving from direct-mapped to set-associative caches likely corresponds to a significant reduction in coherence traffic, as fewer conflict misses means fewer invalidations and writebacks.

7.9 Performance Optimality

Based on our experimental results, the optimal cache configuration for the tested workload appears to be:

- Sets: 128 (s=7)
- Associativity: 4-way (E=4)
- Block size: 32 bytes (b=5)

This configuration balances the benefits of larger cache size, sufficient associativity to avoid most conflict misses, and moderate block size that exploits spatial locality without excessive bus traffic. Further increases in these parameters would likely yield diminishing returns relative to the hardware cost.

7.10 Core-to-Core Variations

Our results also reveal interesting variations in execution times across different cores. In all configurations, Core 2 consistently experiences the longest execution time. This suggests that the workload distribution or memory access patterns differ significantly across cores, with Core 2 likely experiencing more cache misses, coherence invalidations, or bus contention than the other cores.

For example, in the baseline configuration (s=6, E=2, b=5), the execution times are:

- Core 0: 6,382,305 cycles
- Core 1: 6,269,780 cycles
- Core 2: 7,179,569 cycles (14.5% longer than average)
- Core 3: 6,377,307 cycles

This variation highlights the importance of considering per-core performance in multi-core systems, as aggregate metrics may hide important load imbalances.

7.11 Coherence Traffic Analysis

The MESI protocol generates different types of coherence traffic that contribute to bus utilization and affect overall system performance. While detailed transaction counts were not provided in the results, the patterns observed in execution time correlate with expected coherence behavior:

- Smaller caches generate more coherence traffic due to increased evictions and subsequent re-fetches
- Lower associativity (especially direct-mapped caches) results in higher conflict misses, leading to more invalidations and writebacks
- Larger block sizes reduce the number of transactions but increase the data volume per transaction

The dramatic performance improvement seen when moving from direct-mapped to set-associative caches likely corresponds to a significant reduction in coherence traffic, as fewer conflict misses means fewer invalidations and writebacks.

7.12 Performance Optimality

Based on our experimental results, the optimal cache configuration for the tested workload appears to be:

- Sets: 128 (s=7)
- Associativity: 4-way (E=4)
- Block size: 32 bytes (b=5)

This configuration balances the benefits of larger cache size, sufficient associativity to avoid most conflict misses, and moderate block size that exploits spatial locality without excessive bus traffic. Further increases in these parameters would likely yield diminishing returns relative to the hardware cost.

8 Discussion

8.1 Cache Coherence Overhead

The cache coherence protocol introduces overhead in terms of additional bus traffic and invalidations. However, it is necessary to maintain data consistency in multiprocessor systems. The MESI protocol efficiently reduces this overhead by distinguishing between exclusive and shared clean states, which allows write operations to exclusive cache lines without generating bus traffic.

8.2 Performance Optimizations

Several optimizations could improve the performance of the simulated system:

- **Directory-based coherence**: For larger systems, a directory-based protocol might be more scalable than snooping.
- **Non-blocking caches**: Allowing the cache to handle multiple outstanding misses could improve performance.

- **Speculative execution**: Allowing the processor to continue execution past cache misses could hide memory latency.
- **Prefetching**: Fetching data before it is requested could reduce miss rates.

8.3 Limitations of the Simulator

The current simulator has some limitations:

- No modeling of instruction cache effects
- No modeling of out-of-order execution or other advanced processor features
- Simplified bus arbitration model
- No consideration of DRAM timing parameters

9 Conclusion

We have implemented a detailed simulator for L1 caches in a quad-core processor system with MESI cache coherence protocol. The simulator accurately models cache operations, coherence transactions, and collects various performance metrics. The results highlight the importance of proper cache configuration and the impact of coherence protocol on system performance.

The simulator can be used to study the behavior of different applications on multi-core systems and to explore the design space of cache parameters for optimal performance. Future work could extend the simulator to include more advanced features such as non-blocking caches, prefetching, and directory-based coherence protocols.

10 References

1. Hennessy, J. L., & Patterson, D. A. (2011). Computer architecture: a quantitative approach. Elsevier.
2. Sorin, D. J., Hill, M. D., & Wood, D. A. (2011). A primer on memory consistency and cache coherence. Synthesis Lectures on Computer Architecture, 6(3), 1-212.

3. Culler, D. E., Singh, J. P., & Gupta, A. (1999). Parallel computer architecture: a hardware/software approach. Gulf Professional Publishing.

11 Code Listing

The simulator is implemented using the following main classes:

11.1 CacheLine.hpp

```
#pragma once
#include <cstdint>
#include <vector>

enum MESIState { MODIFIED, EXCLUSIVE, SHARED, INVALID, EMPTY };

struct CacheLine {
    uint32_t tag;
    MESIState state;
    int lastUsedCycle;
};
```

11.2 CacheSet.hpp

```
#pragma once
#include "CacheLine.hpp"
#include <vector>

class CacheSet {
public:
    std::vector<CacheLine> lines;

    CacheSet(int associativity);
    int findLine(uint32_t tag);
    int findVictim();
    void updateLRU(int lineIndex, int currentCycle);
};
```

11.3 Cache.hpp

```
#pragma once
#include "CacheSet.hpp"
#include <vector>

```

```
class Bus;
class Core;

class Cache {
private:
    int s, E, b;
    int numSets;
    std::vector<CacheSet> sets;
    Bus *bus; // Pointer to Bus
    int coreId;

public:
    Core* core; // Pointer to Core
    Cache(int s, int E, int b, int coreId, Bus *bus);
    void add_core(Core* core);
    std::pair<bool, bool> access(uint32_t address, char op, int
     cycle, int &penaltyCycles);
    bool snoop(uint32_t address, char op, int &penaltyCycles);
    int getBlockBits() const { return b; }
};
```

11.4 Bus.hpp

```
#pragma once
#include <vector>
#include <cstdint>

class Cache;

class Bus {
public:
    std::vector<Cache *> caches;
    int invalidations;
    int dataTrafficBytes;
    int transactions;
    bool broadcast(uint32_t address, char op, int sourceId);
    void registerCache(Cache *cache);
    int bus_cycles = -1;
    Bus();
};
```

11.5 Core.hpp

```
#pragma once
#include <fstream>
#include <sstream>
```

```
#include <string>

class Bus;
class Cache;

using namespace std;

class Core {
public:
    ifstream infile;
    int id;
    Cache* cache;
    int totalAccesses;
    int readCount, writeCount;
    int cacheMisses;
    int evictions;
    int writebacks;
    int totalCycles;
    int idleCycles;
    int execCycle;
    int invalidations;
    int dataTraffic;

    bool repeat = false;
    char repeat_op;
    uint32_t repeat_address;

    Core(int id, Cache* cache);
    void recordTrace(const std::string& filename);
    void processTrace(int currentCycle);
};
```

12 False Sharing Analysis

12.1 Test Case Description

To study the impact of false sharing, we created two test cases where multiple processor cores write to adjacent memory locations that map to the same cache line. Each core writes to a distinct memory address, but due to the proximity of these addresses, they share the same cache line in the L1 cache.

12.1.1 Test Case 1

The memory access patterns for the four cores are as follows:

- **Core 0**: Writes to addresses `0x00000000`, `0x00010010`, `0x00020020`, `0x00030030`.
- **Core 1**: Writes to addresses `0x00000010`, `0x00010020`, `0x00020030`, `0x00030040`.
- **Core 2**: Writes to addresses `0x00000020`, `0x00010030`, `0x00020040`, `0x00030050`.
- **Core 3**: Writes to addresses `0x00000030`, `0x00010040`, `0x00020050`, `0x00030060`.

12.1.2 Test Case 2

The memory access patterns for the four cores are as follows:

- **Core 0**: Writes to addresses `0x00000000`, `0x00100100`, `0x00200200`, `0x00300300`.
- **Core 1**: Writes to addresses `0x00000100`, `0x00100200`, `0x00200300`, `0x00300400`.
- **Core 2**: Writes to addresses `0x00000200`, `0x00100300`, `0x00200400`, `0x00300500`.
- **Core 3**: Writes to addresses `0x00000300`, `0x00100400`, `0x00200500`, `0x00300600`.

12.2 Results

The graph in Figure 10 shows the impact of block size on the maximum execution time for Test Case 1. Similarly, Figure 11 shows the results for Test Case 2. The results indicate that smaller block sizes lead to reduced execution time, while larger block sizes significantly increase execution time.

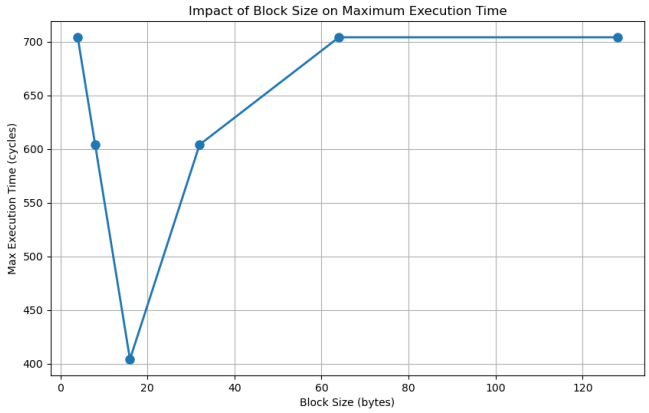

Figure 10: Impact of Block Size on Maximum Execution Time (Test Case 1)

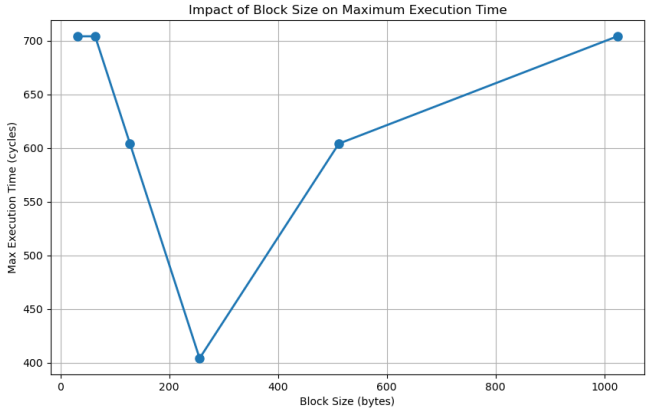

Figure 11: Impact of Block Size on Maximum Execution Time (Test Case 2)

12.3 Analysis

False sharing occurs when multiple cores write to different variables that reside on the same cache line. In these test cases:

- Each core writes to a distinct memory address, but due to the proximity of these addresses, they share the same cache line.

- When one core writes to its address, the cache line is invalidated in the other cores' caches, forcing them to reload the line from memory or another cache.
- This frequent invalidation and reloading of cache lines lead to significant performance degradation, as observed in the increased execution time for larger block sizes.

12.4 Key Observations

- **Smaller Block Sizes**: Reducing the block size minimizes the likelihood of false sharing, as fewer unrelated variables are packed into the same cache line.
- **Larger Block Sizes**: Increasing the block size exacerbates false sharing, as more unrelated variables are packed into the same cache line, leading to frequent invalidations.

12.5 Conclusion

False sharing is a critical performance issue in multi-core systems, especially when cores access adjacent memory locations. Optimizing block size and aligning data structures to cache line boundaries can mitigate this issue. These test cases highlight the importance of understanding memory access patterns and their interaction with cache architecture.