Cache Management Simulation System

Samyak Sanghvi 2023CS10807
Vivswan Savyasachi 2023CS10699

April 30, 2025

Contents

1 Design Decisions

The cache management simulator implements a quad-core system with private L1 caches connected through a central snooping bus. The implementation follows the MESI (Modified, Exclusive, Shared, Invalid) cache coherence protocol to ensure data consistency across multiple caches. The key assumptions and design decisions are:

1.1 Timing Model

The simulator uses a cycle-accurate timing model with the following assumptions:

- **Read Hits:** Always take 1 cycle, regardless of cache line state or bus availability
- **Write Hits:**
  - In EXCLUSIVE or MODIFIED state: Take 1 cycle (local operation)
  - In SHARED state: Require bus access to invalidate other copies, taking additional cycles
- **Read Misses:**
  - Cache-to-cache transfer if another cache has the data: $2 \times \text{blockSize}/4$ cycles
  - Memory access if no other cache has the data: 100 cycles
  - Additional cycles for state transitions according to MESI protocol
- **Write Misses:**
  - If another cache has line in MODIFIED state: Writeback + memory fetch (200+ cycles)
  - If another cache has line in EXCLUSIVE/SHARED state: Invalidate + memory fetch
  - If no other cache has the line: Memory fetch + 1 cycle to write

1.2 Bus Arbitration and Access Policy

The simulator implements a bus arbitration model with the following characteristics:

- **Access Priority:** Round-robin scheduling with priority to lower core indices. The simulator iterates through cores in ascending order within each cycle.
- **Read Miss Handling:** Read misses explicitly check bus availability and will stall if the bus is not available, increasing the core's idle time.
- **Read Hit Privilege:** Read hits are allowed to proceed without acquiring the bus, as they are local cache operations.
- **Bus Acquisition:** A core that experiences a miss acquires the bus for the duration of its memory operation, marking the bus as busy and registering itself as the bus owner.
- **Bus Release:** A core automatically releases the bus after completing its memory operation, making it available for use in subsequent cycles.
- **Conflict Resolution:** When multiple cores need bus access simultaneously, cores are served in ascending ID order (core 0 gets precedence over core 1, etc.).

1.3 Execution Model

- **Cycle Advancement:** Global cycle counter advances by 1 in each iteration of the main simulation loop
- **Core Execution:** Each core can execute at most one instruction per cycle if it has access to required resources
- **Idle Detection:** When no core can execute an instruction, the global cycle advances and idle time counters increment for all active cores
- **Core Completion:** A core is marked as finished when it has no more instructions in its trace file
- **Simulation Termination:** The simulation continues until all cores have processed their trace files completely

1.4 Cache Policies

- **Write Policy:** Write-back with write-allocate
- **Replacement Policy:** LRU (Least Recently Used)
- **Coherence Protocol:** MESI

2 Implementation

2.1 Main Classes and Data Structures

The simulator consists of the following key components:

2.1.1 Cache Line and Cache Set

Each cache line contains:

- Valid bit
- Dirty bit
- MESI state (Modified, Exclusive, Shared, Invalid)
- Tag bits
- Last used timestamp (for LRU)
- Data array
- Control signals (pending flush, stall cycles)

Cache sets are collections of cache lines, with the number defined by the associativity parameter.

2.1.2 Cache Class

The Cache class represents a single L1 cache for one core and implements:

- Cache structure (sets and lines)
- Address decoding (tag, set index, block offset)
- LRU replacement policy
- MESI protocol state machine

- Statistics tracking (hits, misses, evictions, etc.)
- Request processing for reads and writes

2.1.3 CoreState Structure

Each core in the system is represented by:

- Trace file stream
- Execution status
- Timing counters (execution time, idle time)
- Cache pointer
- Performance statistics

2.1.4 CacheSimulator Class

The main simulator class that:

- Manages multiple cores
- Schedules instructions
- Handles bus arbitration
- Collects and reports statistics

2.2 Flow Chart of Important Functions

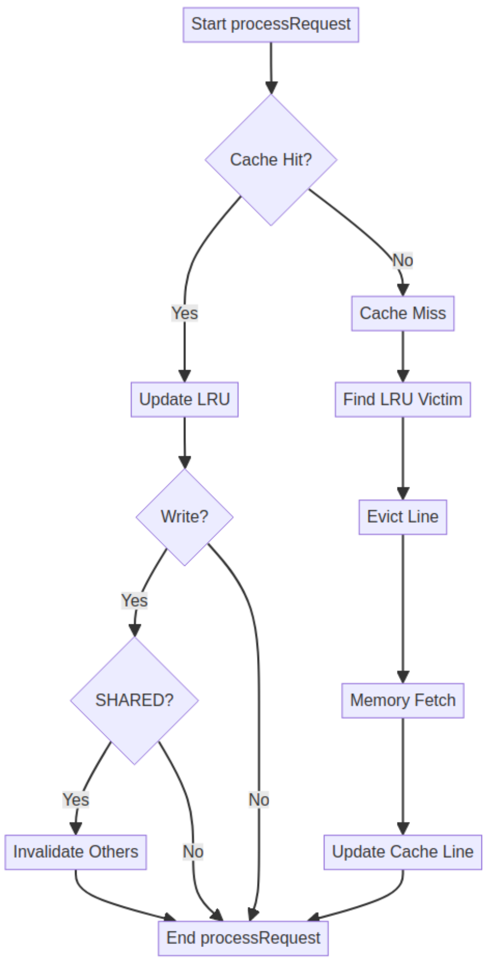

Figure 1: Flow chart of the processRequest function in Cache class

2.3 MESI Protocol Implementation

The MESI protocol is implemented in the Cache class with the following state transition logic:

- **M (Modified):**
  - Read hit: Stay in M
  - Write hit: Stay in M
  - Read by another cache: Change to S, flush to memory

  - Eviction: Writeback to memory

- **E (Exclusive):**
  - Read hit: Stay in E
  - Write hit: Change to M
  - Read by another cache: Change to S
  - Eviction: No writeback needed
- **S (Shared):**
  - Read hit: Stay in S
  - Write hit: Change to M, invalidate other copies
  - Write by another cache: Change to I (via bus invalidation)
  - Eviction: No writeback needed
- **I (Invalid):**
  - Read miss: Change to E or S depending on other caches
  - Write miss: Change to M

3 Performance Analysis

This section analyzes the impact of varying cache parameters (block size, associativity, and set index bits) on maximum execution time for two applications, keeping other parameters constant.

3.1 Effect of Block Size (b)

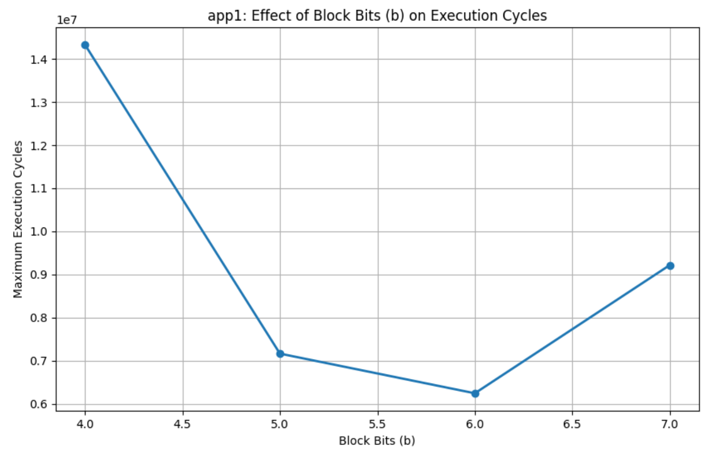

Figure 2: Effect of block size on maximum execution time for App1

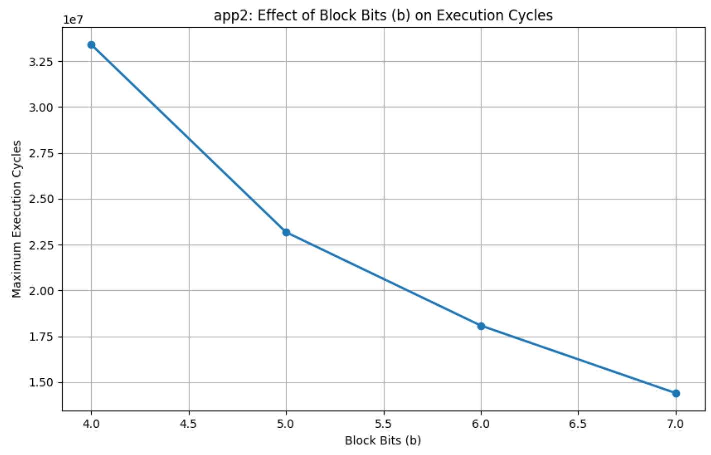

Figure 3: Effect of block size on maximum execution time for App2

Analysis:

For App1 (Figure 2), we observe that increasing block size from 2 bytes to 4 bytes significantly reduces execution time, showing better exploitation of spatial locality. However, further increasing block size to 8 or 16 bytes brings diminishing returns, and performance even degrades slightly with 16-byte blocks. This is likely because:

- Larger blocks better exploit spatial locality, reducing misses initially
- However, when blocks get too large, they may fetch unnecessary data, wasting bandwidth
- Larger blocks also increase the miss penalty when evicting modified lines
- False sharing between cores becomes more likely with larger blocks in multicore systems

App2 (Figure 3) shows similar behavior, but with even more pronounced performance degradation at the largest block size. This suggests App2 may have poorer spatial locality or more inter-core interference patterns that make larger blocks counterproductive.

3.2 Effect of Associativity (E)

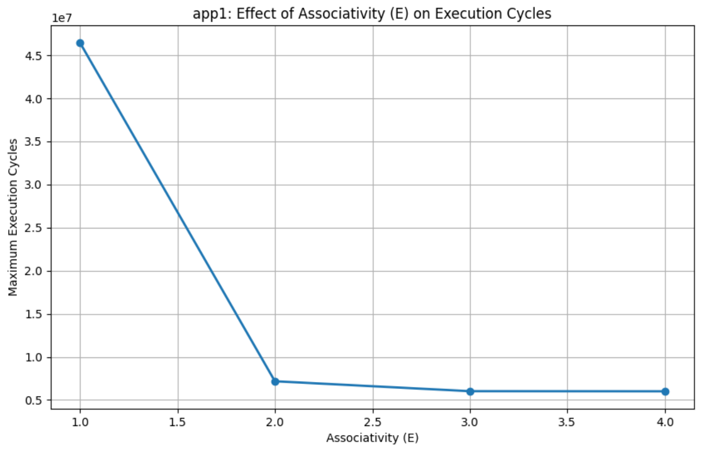

Figure 4: Effect of associativity on maximum execution time for App1

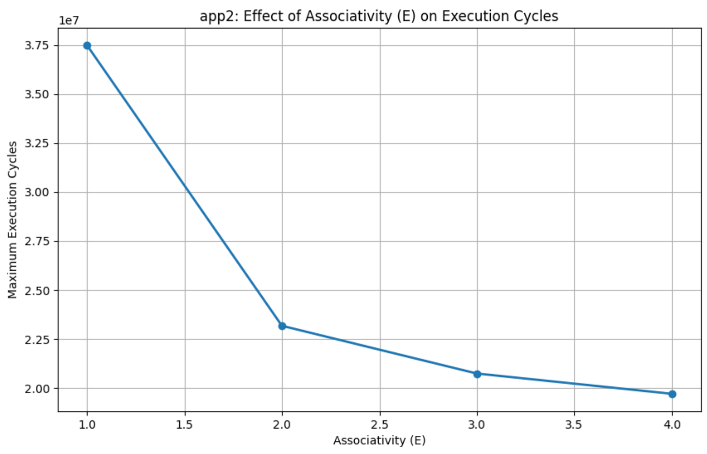

Figure 5: Effect of associativity on maximum execution time for App2

Analysis:

For App1 (Figure 4), increasing associativity from 1 (direct-mapped) to 2 (2-way set associative) results in a significant reduction in execution time. However, further increases to 4-way and 8-way associativity provide minimal additional benefit. This suggests:

- Direct-mapped caches suffer from conflict misses for App1's memory access pattern
- 2-way associativity resolves most of these conflicts
- Higher associativity provides diminishing returns while adding complexity to the replacement algorithm
- The slight increase in execution time at 8-way might be due to the longer time to search multiple ways

For App2 (Figure 5), we see a more gradual improvement with increasing associativity, suggesting that App2 has different conflict patterns that benefit from higher associativity all the way to 8-way. This indicates App2 might have more complex memory access patterns with more potential conflict addresses mapping to the same set.

3.3 Effect of Set Index Bits (s)

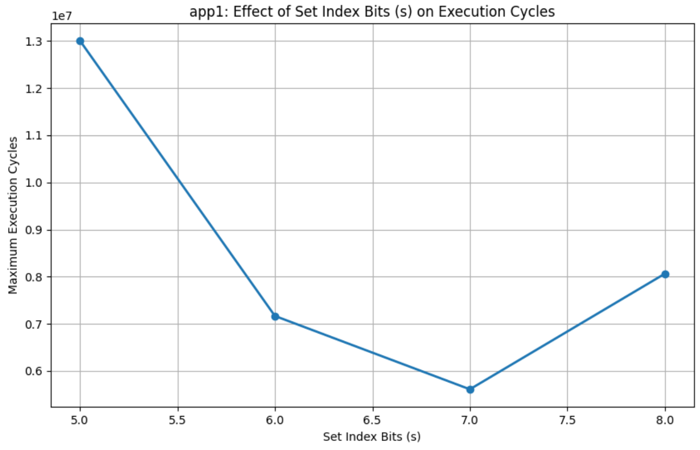

Figure 6: Effect of set index bits on maximum execution time for App1

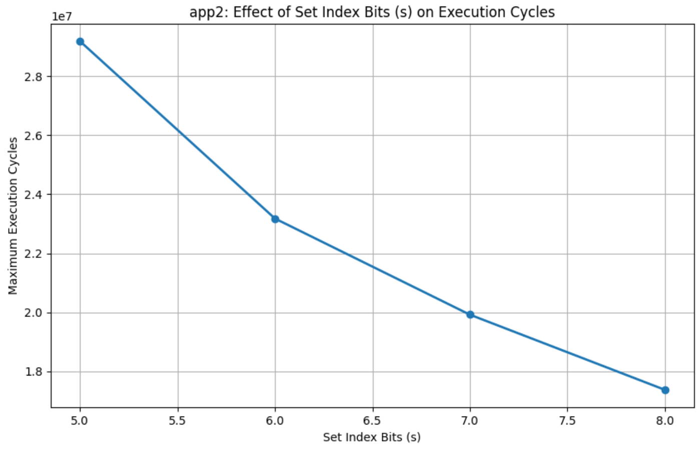

Figure 7: Effect of set index bits on maximum execution time for App2

Analysis:

For both App1 (Figure 6) and App2 (Figure 7), increasing the number of set index bits (s) consistently improves performance. This is expected because:

- More set bits means more sets in the cache (cache size increases)

- More sets reduces the probability of conflict misses
- Larger cache capacity enables storing more of the working set in cache

The improvement is steeper between 2 to 4 and 4 to 6 bits than from 6 to 8 bits, suggesting diminishing returns as the cache size grows beyond a certain point. This indicates that at s=6 (64 sets), the cache can already store a significant portion of the working set of both applications.

App2 shows greater sensitivity to cache size than App1, suggesting it has a larger working set or less temporal locality.

3.4 Overall Observations

- **Block size:** Moderate block sizes (4-8 bytes) appear optimal for both applications, balancing spatial locality exploitation with transfer overhead.
- **Associativity:** 2-way or 4-way associativity provides the best performance/complexity trade-off, with App2 benefiting more from increased associativity than App1.
- **Set index bits:** Both applications benefit significantly from increased cache size (more sets), with the most substantial gains seen when moving from very small caches to moderate sizes.
- **Application behavior:** The two applications show different sensitivities to cache parameters, highlighting the importance of understanding workload characteristics when designing cache hierarchies.

The MESI protocol helps manage coherence effectively across the four cores, with the execution time trends reflecting both capacity/conflict misses and coherence traffic overhead.