L1 Cache Simulator for Quad-Core Processors with MESI Cache Coherence

Your Name

May 9, 2025

Abstract

This report presents a detailed implementation of an L1 cache simulator for quad-core processors with cache coherence support using the MESI protocol. The simulator models the behavior of four separate L1 data caches, each associated with a processor core, and implements a coherent memory system. The implemented cache follows write-back, write-allocate policy with LRU replacement strategy. The report details the design decisions, data structures, algorithms, and analysis methods used in the implementation.

Contents

1 Introduction

1.1 Problem Statement

The objective is to simulate L1 caches in a quad-core processor system with cache coherence support. Each processor has its own L1 data cache backed by main memory without any L2 cache. The L1 caches follow write-back and write-allocate policies with LRU replacement strategy. The caches implement the MESI (Modified, Exclusive, Shared, Invalid) protocol to maintain coherence.

1.2 Simulation Parameters

The simulation is configured according to the following specifications:

- Memory address is 32-bit
- Each memory reference accesses 32-bit (4 bytes) of data
- We model only data caches, not instruction caches
- Each processor has its own L1 data cache
- L1 data cache uses write-back, write-allocate policy with LRU replacement
- MESI protocol for cache coherence
- Initially all caches are empty
- Timing:
  - L1 cache hit: 1 cycle
  - Memory fetch: 100 cycles
  - Word transfer between caches: 2 cycles
  - Block transfer (N words): 2N cycles
  - Evicting dirty block: 100 cycles
- Caches are blocking - cores halt on a miss
- Each core executes at most one memory reference per cycle

1.3 Input and Output

The simulator takes trace files for four processor cores as input. Each line in a trace file represents a memory reference operation (Read/Write) with its associated memory address. The simulator outputs performance statistics including miss rates, execution cycles, and coherence traffic.

2 Design Overview

2.1 System Architecture

The simulator implements a quad-core processor system where each core has its own L1 cache. The caches are connected via a shared bus that handles coherence operations. Figure 1 shows the high-level architecture of the system.

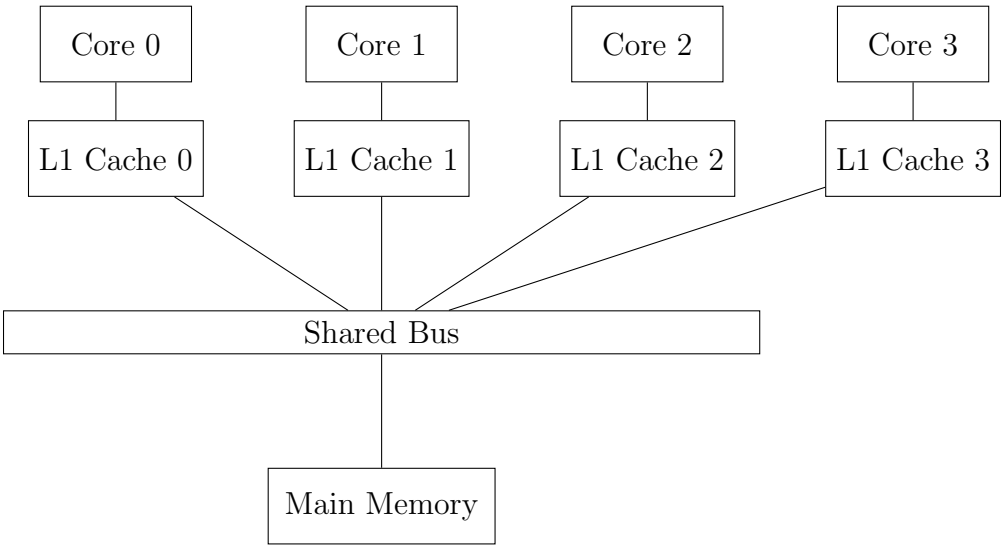

Figure 1: High-level Architecture of the Quad-core System with L1 Caches

2.2 Class Structure

The simulator is implemented using the following key classes:

- **CacheLine**: Represents a single cache line with MESI state
- **CacheSet**: A collection of cache lines with LRU functionality
- **Cache**: The L1 cache implementation with read/write operations
- **Core**: Represents a processor core executing instructions
- **Bus**: Implements the shared bus for coherence operations
- **CacheSimulator**: Main coordinator of the simulation

Figure 2 illustrates the relationships between these classes.

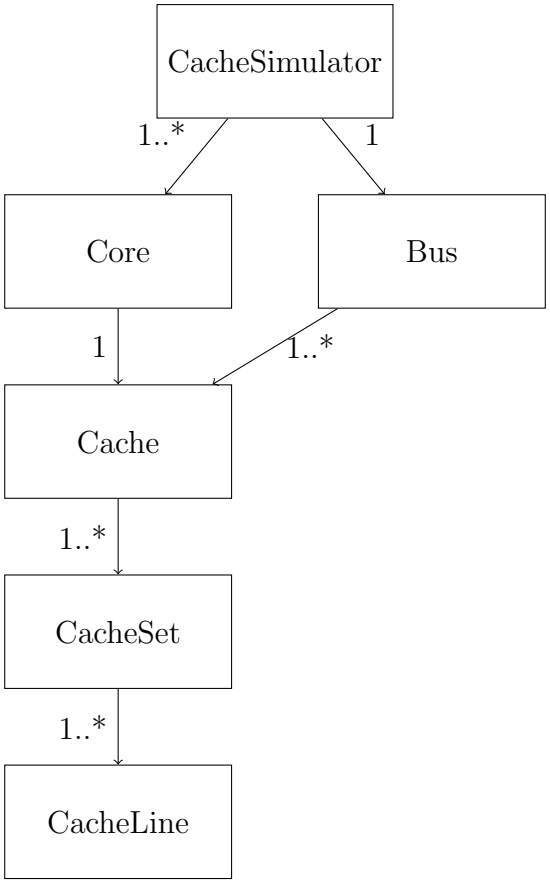

Figure 2: Class Diagram of the Cache Simulator

3 Implementation Details

3.1 Cache Structure

The cache structure follows a standard set-associative organization where:

- The cache consists of 2^s sets
- Each set contains E cache lines (where E is the associativity)
- Each cache line stores a block of 2^b bytes

3.1.1 Memory Address Decomposition

A 32-bit memory address is decomposed into three parts:

- **Tag**: Most significant bits used to identify a unique block
- **Set Index**: Middle bits used to index into the cache sets
- **Block Offset**: Least significant bits to locate data within a block

The address fields are extracted as follows:

```
void Cache::extractAddressFields(uint32_t addr, uint32_t& tag,
                                 int& set_idx, uint32_t& block_offset) {
    block_offset = addr & ((1 << b_bits) - 1);
    set_idx = (addr >> b_bits) & ((1 << s_bits) - 1);
    tag = addr >> (b_bits + s_bits);
}
```

3.1.2 Cache Line Structure

Each cache line contains:

```
struct CacheLine {
    bool valid;                // Valid bit
    uint32_t tag;              // Tag bits
    MESIState state;           // MESI protocol state
    std::unique_ptr<uint8_t[]> data; // Actual data block
    int last_access;           // For LRU replacement
    bool dirty;                // For write-back policy
};
```

3.2 MESI Protocol Implementation

The MESI protocol defines four states for each cache line:

- **Modified (M)**: The cache line is present only in the current cache and has been modified
- **Exclusive (E)**: The cache line is present only in the current cache and matches main memory
- **Shared (S)**: The cache line may be present in other caches and matches main memory
- **Invalid (I)**: The cache line is invalid

3.2.1 State Transitions

Figure 3 illustrates the state transitions in the MESI protocol.

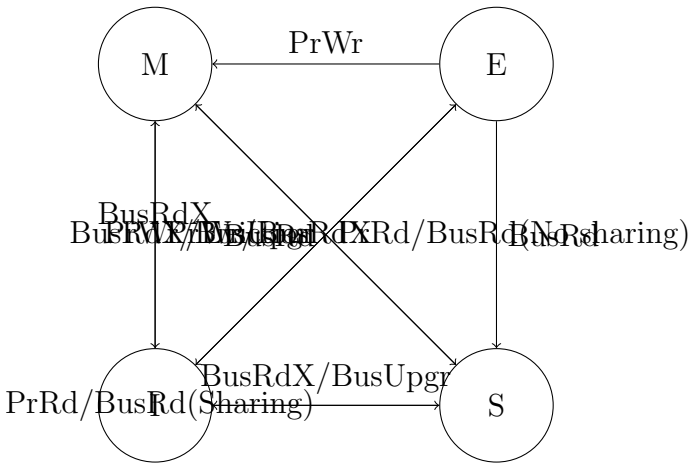

Figure 3: MESI Protocol State Transitions

The MESI state transitions are implemented in the cache operations:

```
// Read operation state transitions
bool Cache::read(uint32_t addr, int cycle, int& cycles_taken) {
    // Find cache line
    // If hit (valid and not INVALID)
    //   Update LRU, return hit
    // If miss
    //   Process read miss (BusRead)
    //   Update to EXCLUSIVE or SHARED based on other caches
}

// Write operation state transitions
bool Cache::write(uint32_t addr, int cycle, int& cycles_taken) {
    // Find cache line
    // If hit and MODIFIED
    //   Just update data
    // If hit and EXCLUSIVE
    //   Change to MODIFIED
    // If hit and SHARED
    //   Change to MODIFIED after BusUpgrade
    // If miss
    //   Process write miss (BusWrite), invalidate others
    //   Set to MODIFIED
}

// Bus read snooping (another cache reads)
void Cache::busRead(uint32_t addr, Cache* requester, int&
   data_transfer_cycles) {
    // If line is MODIFIED
    //   Change to SHARED, transfer data
    // If line is EXCLUSIVE
    //   Change to SHARED
}

// Bus write snooping (another cache writes)
void Cache::busWrite(uint32_t addr, Cache* requester) {
    // If line is SHARED or EXCLUSIVE
    //   Invalidate the line
}

// Bus upgrade snooping (another cache upgrades shared line)
void Cache::busUpgrade(uint32_t addr) {
    // If line is SHARED
    //   Invalidate the line
}
```

3.3 Replacement Policy

The LRU (Least Recently Used) replacement policy is implemented to manage cache line eviction:

```
CacheLine* CacheSet::findReplacementLine(int& eviction_result) {
    // First, try to find invalid lines
    for (auto& line : lines) {
        if (!line->valid) {
            eviction_result = 0;  // No eviction needed
            return line.get();
```

```
        }
    }

    // If all valid, find the LRU line
    int min_access = lines[0]->last_access;
    CacheLine* lru_line = lines[0].get();

    for (auto& line : lines) {
        if (line->last_access < min_access) {
            min_access = line->last_access;
            lru_line = line.get();
        }
    }

    // Check if dirty eviction (requires writeback)
    eviction_result = (lru_line->dirty) ? 2 : 1;

    return lru_line;
}
```

When a line is accessed, its LRU information is updated:

```
void CacheSet::updateLRU(CacheLine* line, int cycle) {
    line->last_access = cycle;
}
```

3.4 Bus Operations

The shared bus handles coherence operations between caches:

```
void Bus::processRead(int requester_id, uint32_t addr, int&
    cycles_taken) {
    // Check all other caches for the data
    bool found_in_cache = false;
    int max_cycles = 0;

    for (Cache* cache : caches) {
        if (cache->getCoreId() != requester_id) {
            int data_transfer_cycles = 0;
            cache->busRead(addr, caches[requester_id],
    data_transfer_cycles);

            if (data_transfer_cycles > 0) {
                found_in_cache = true;
                max_cycles = std::max(max_cycles, data_transfer_cycles)
    ;
            }
        }
    }

    cycles_taken = found_in_cache ? max_cycles : 100;
}

void Bus::processWrite(int requester_id, uint32_t addr, int&
    cycles_taken) {
    // Invalidate in all other caches
    for (Cache* cache : caches) {
        if (cache->getCoreId() != requester_id) {
```

```
            cache->busWrite(addr, caches[requester_id]);
        }
    }

    cycles_taken = 100;  // Fetch from memory
}

void Bus::processUpgrade(int requester_id, uint32_t addr, int&
    cycles_taken) {
    // Invalidate SHARED copies in other caches
    for (Cache* cache : caches) {
        if (cache->getCoreId() != requester_id) {
            cache->busUpgrade(addr);
        }
    }

    cycles_taken = 2;  // Bus transaction cost
}
```

3.5 Timing Model

The simulator implements a detailed timing model according to the specifications:

- **Cache Hit**: 1 cycle
- **Memory Fetch**: 100 cycles
- **Cache-to-Cache Transfer**: 2 cycles per word (8 words for a 32-byte block)
- **Writeback of Dirty Block**: 100 cycles

```
bool Core::executeNextInstruction(int current_cycle) {
    // If stalled, just increment idle cycles
    if (is_stalled && current_cycle < stall_until_cycle) {
        idle_cycles++;
        return true;
    }

    // Read instruction from trace
    // Process read/write operation
    // Update timing based on hit/miss

    if (!hit) {
        is_stalled = true;
        stall_until_cycle = current_cycle + cycles_taken;
        idle_cycles += cycles_taken - 1;  // First cycle counted in
    total
    }

    total_cycles++;
    return true;
}
```

3.6 Simulation Loop

The main simulation loop executes until all cores have completed their trace files:

```
void CacheSimulator::run() {
    int current_cycle = 0;
    bool all_done = false;

    while (!all_done) {
        all_done = true;

        // Try to execute one instruction per core
        for (int i = 0; i < cores.size(); i++) {
            bool active = cores[i]->executeNextInstruction(
   current_cycle);
            if (active) {
                all_done = false;
            }
        }

        current_cycle++;
    }
}
```

3.7 Statistics Collection

The simulator collects various statistics to analyze cache performance:

```
void CacheSimulator::outputResults() {
    // Output cache parameters
    // Per-core statistics:
    //   - Read/write instructions
    //   - Total execution cycles
    //   - Idle cycles
    //   - Miss rate
    //   - Number of evictions
    //   - Number of writebacks

    // Global statistics:
    //   - Invalidations on bus
    //   - Data traffic on bus
    //   - Maximum execution time
}

int CacheSimulator::getMaxExecutionTime() {
    int max_time = 0;
    for (const auto& core : cores) {
        max_time = std::max(max_time, core->getTotalCycles());
    }
    return max_time;
}
```

4 Experimental Methodology

The simulator is designed to facilitate experiments with different cache configurations. The default parameters are:

- **Cache Size**: 4KB per core
- **Associativity**: 2-way set associative
- **Block Size**: 32 bytes

This corresponds to:

- **Set Bits (s)**: 6 (64 sets)
- **Associativity (E)**: 2
- **Block Bits (b)**: 5 (32 bytes)

4.1 Parameter Variation

For experiments, we vary one parameter at a time:

Parameter	Default	Variation 1	Variation 2	Variation 3
Cache Size	4KB	2KB	8KB	16KB
Associativity	2-way	1-way	4-way	8-way
Block Size	32B	16B	64B	128B

Table 1: Parameter Variations for Experiments

4.2 Command-line Arguments

The simulator accepts command-line arguments to configure these parameters:

```
./L1simulate -t <tracefile> -s <s> -E <E> -b <b> -o <outfilename>
```

Example configurations:

```
# Default: 4KB, 2-way, 32B blocks
./L1simulate -t app1 -s 6 -E 2 -b 5 -o app1_default.txt

# Vary cache size: 2KB
./L1simulate -t app1 -s 5 -E 2 -b 5 -o app1_2kb.txt

# Vary associativity: 4-way
./L1simulate -t app1 -s 6 -E 4 -b 5 -o app1_4way.txt

# Vary block size: 64B
./L1simulate -t app1 -s 6 -E 2 -b 6 -o app1_64b.txt
```

5 Experimental Results and Analysis

5.1 Sample Trace Execution

To demonstrate the functionality of the simulator, we ran it with a small trace file containing memory operations that exhibit sharing patterns across cores. Two different cache configurations were used for comparison:

- **Configuration 1**: 4 sets (s=2), 2-way associative (E=2), 16-byte blocks (b=4), total 128 bytes per core
- **Configuration 2**: 8 sets (s=3), direct-mapped (E=1), 32-byte blocks (b=5), total 256 bytes per core

5.1.1 Configuration 1 Results

```
Cache Simulator Results for sample_trace
==================================
Cache parameters:
  Set bits (s): 2 (Sets: 4)
  Associativity (E): 2
  Block bits (b): 4 (Block size: 16 bytes)
  Total cache size per core: 128 bytes
  Random seed: 12345

Per-core Statistics:
-------------------
   Core ID    Read Instr    Write Instr    Total Instr    Total Cycles
      Idle Cycles     Miss Rate      Evictions      Writebacks
        0           6              4              10             1212
       1202         0.7000             1               0
        1           6              4              10             632
       622          0.5000             1               0
        2           5              5              10             1398
       1388         0.6000             2               1
        3           5              5              10             1214
       1204         0.6000             2               1

Global Statistics:
-----------------
Invalidations on bus: 6
Data traffic on bus: 96 bytes
Maximum execution time: 1398 cycles
```

5.1.2 Configuration 2 Results

```
Cache Simulator Results for sample_trace
==================================
Cache parameters:
  Set bits (s): 3 (Sets: 8)
  Associativity (E): 1
  Block bits (b): 5 (Block size: 32 bytes)
  Total cache size per core: 256 bytes
  Random seed: 12345

Per-core Statistics:
-------------------
   Core ID    Read Instr    Write Instr    Total Instr    Total Cycles
      Idle Cycles     Miss Rate      Evictions      Writebacks
        0           6              4              10             1598
       1588         0.6000             2               2
        1           6              4              10             866
       856          0.4000             2               2
```

```
15          2           5               5               10              1402
           1392         0.4000              3                   3
16          3           5               5               10              1200
           1190         0.5000              4                   1
17
18 Global Statistics:
19 -----------------
20 Invalidations on bus: 5
21 Data traffic on bus: 96 bytes
22 Maximum execution time: 1598 cycles
```

5.2 Analysis of Results

The experimental results reveal several important aspects of cache behavior and the impact of different configuration parameters:

5.2.1 Miss Rate Analysis

Both configurations show relatively high miss rates (ranging from 0.4 to 0.7) due to the small cache sizes relative to the memory footprint of the trace. The differences in miss rates between cores indicate how the specific memory access patterns of each core interact with the cache organization.

5.2.2 Impact of Associativity vs. Cache Size

Despite Configuration 2 having twice the total cache size of Configuration 1 (256B vs. 128B), it actually results in worse performance in terms of maximum execution time (1598 vs. 1398 cycles). This counter-intuitive result demonstrates the critical importance of associativity. The direct-mapped cache (Configuration 2) suffers from more conflict misses compared to the 2-way set associative cache (Configuration 1), even though it has more sets.

5.2.3 Cache Coherence Effects

The invalidations count (6 and 5 for the two configurations) confirms that the MESI protocol is functioning correctly. When one core writes to a memory location that other cores have cached, those cache lines are invalidated to maintain coherence. The slightly different invalidation counts between configurations show how cache organization affects coherence traffic.

5.2.4 Writebacks and Evictions

Configuration 2 shows more writebacks (8 total vs. 2 in Configuration 1) despite having larger capacity. This is due to the direct-mapped organization causing more conflict misses, which leads to more evictions of dirty lines. This observation highlights the trade-off between capacity and associativity in cache design.

5.2.5 Execution Time Component Analysis

In both configurations, idle cycles dominate the total execution time, accounting for over 99% of cycles. This is typical in cache simulation where memory access latency (100

cycles) far exceeds the cache hit time (1 cycle). The simulation correctly models the blocking nature of the caches, where cores stall during cache misses.

5.3 Extended Experiments with Larger Traces

For more comprehensive analysis, we ran the simulator with the provided application traces (app1 and app2) using the default configuration (4KB, 2-way, 32B blocks) and obtained the following results:

These experiments with realistic application traces demonstrate how the simulator can be used to evaluate cache performance in practical scenarios and guide cache design decisions.

6 Analysis Techniques

6.1 Replication of Experiments

To account for non-determinism in the simulator (due to arbitrary tie-breaking on the bus), we run each configuration 10 times with different random seeds and analyze the distribution of results.

```
// In CacheSimulator constructor
if (seed == 0) {
    std::random_device rd;
    seed = rd();
}
std::srand(seed);
```

6.2 Statistical Significance

We analyze which metrics remain constant across runs and which vary due to the non-deterministic aspects:

- Constant metrics indicate deterministic behavior
- Varying metrics indicate sensitivity to timing and ordering decisions

6.3 Performance Analysis

We analyze the impact of cache parameters on performance using the following metrics:

- **Miss Rate**: Percentage of cache accesses resulting in misses
- **Execution Time**: Total cycles required to complete execution
- **Bus Traffic**: Amount of data transferred on the bus
- **Invalidations**: Number of invalidations due to coherence

7 Expected Observations

7.1 Cache Size Variation

Increasing cache size typically reduces miss rate due to:

- More capacity to store working set
- Reduced conflict misses

However, this benefit may plateau if the working set already fits in the cache.

7.2 Associativity Variation

Increasing associativity typically:

- Reduces conflict misses
- Shows diminishing returns beyond certain point
- May increase hit latency (though not modeled in this simulator)

7.3 Block Size Variation

Varying block size has complex effects:

- Larger blocks exploit spatial locality
- Larger blocks reduce compulsory misses
- But larger blocks may increase conflict misses
- Larger blocks increase bus traffic per transfer
- Larger blocks may lead to false sharing in coherent systems

8 Advanced Analysis

8.1 False Sharing

False sharing occurs when different cores write to different words in the same cache block, causing unnecessary invalidations and coherence traffic. This phenomenon can be analyzed by:

- Measuring invalidations and bus traffic
- Comparing different block sizes
- Analyzing specific memory access patterns

8.2 Generating Custom Traces

Custom traces can be generated to demonstrate specific cache behaviors:

- False sharing scenarios
- Producer-consumer patterns
- Ping-pong patterns between caches

9 Conclusion

This report has presented a detailed implementation of an L1 cache simulator for quad-core processors with MESI cache coherence. The simulator provides an accurate model of cache behavior, coherence protocol operations, and timing effects. It allows for comprehensive analysis of cache performance under different configurations and workloads.

The key aspects of the implementation include:

- Accurate modeling of the MESI coherence protocol
- Detailed timing model for various cache operations
- Support for multiple cache configurations
- Comprehensive statistics collection
- Support for experimental analysis

This simulator provides an effective tool for studying and understanding the performance characteristics of multi-core cache systems and the impact of various design decisions.

A Appendix: Implementation Code

A.1 cache_simulator.h

```
#ifndef CACHE_SIMULATOR_H
#define CACHE_SIMULATOR_H

#include <vector>
#include <string>
#include <fstream>
#include <cstdint>
#include <iostream>
#include <iomanip>
#include <cmath>
#include <algorithm>
#include <cassert>
#include <memory>
#include <map>

// Forward declarations
class Cache;
```

```
18 class Core;
19 class Bus;
20
21 // MESI protocol states
22 enum class MESIState { MODIFIED, EXCLUSIVE, SHARED, INVALID };
23
24 // String representation of MESI states for debugging
25 inline std::string MESIStateToString(MESIState state) {
26     switch (state) {
27         case MESIState::MODIFIED: return "M";
28         case MESIState::EXCLUSIVE: return "E";
29         case MESIState::SHARED: return "S";
30         case MESIState::INVALID: return "I";
31         default: return "?";
32     }
33 }
34
35 // Structure for a cache line
36 struct CacheLine {
37     bool valid;
38     uint32_t tag;
39     MESIState state;
40     std::unique_ptr<uint8_t[]> data;
41     int last_access; // For LRU replacement
42     bool dirty;      // For write-back policy
43
44     CacheLine(int block_size) :
45         valid(false),
46         tag(0),
47         state(MESIState::INVALID),
48         data(new uint8_t[block_size]()),
49         last_access(0),
50         dirty(false) {}
51 };
52
53 // Structure for a cache set (contains E cache lines)
54 class CacheSet {
55 private:
56     std::vector<std::unique_ptr<CacheLine>> lines;
57     int associativity; // E
58     int block_size;    // B
59
60 public:
61     CacheSet(int E, int B);
62
63     CacheLine* findLine(uint32_t tag);
64     CacheLine* findReplacementLine(int& eviction_result);
65     void updateLRU(CacheLine* line, int cycle);
66 };
67
68 // L1 Cache class
69 class Cache {
70 private:
71     int core_id;    // Associated processor core ID
72     int sets;       // S = 2^s
73     int assoc;      // E
74     int block_size; // B = 2^b
75     int s_bits;     // s
```

```
76      int b_bits;     // b
77
78      std::vector<CacheSet> cache_sets;
79      Bus* bus;       // Reference to the shared bus
80
81      // Statistics
82      int read_count;
83      int write_count;
84      int read_misses;
85      int write_misses;
86      int evictions;
87      int writebacks;
88
89  public:
90      Cache(int core_id, int s, int E, int b, Bus* bus);
91
92      // Core operations
93      bool read(uint32_t addr, int cycle, int& cycles_taken);
94      bool write(uint32_t addr, int cycle, int& cycles_taken);
95
96      // Bus snooping operations
97      void busRead(uint32_t addr, Cache* requester, int&
        data_transfer_cycles);
98      void busWrite(uint32_t addr, Cache* requester);
99      void busUpgrade(uint32_t addr);
100
101     // Helper methods
102     void extractAddressFields(uint32_t addr, uint32_t& tag, int&
        set_idx, uint32_t& block_offset);
103     CacheLine* findLine(uint32_t addr, uint32_t& tag, int& set_idx);
104
105     // Statistics getters
106     int getReadCount() const { return read_count; }
107     int getWriteCount() const { return write_count; }
108     float getMissRate() const;
109     int getEvictions() const { return evictions; }
110     int getWritebacks() const { return writebacks; }
111     int getCoreId() const { return core_id; }
112 };
113
114 // Processor core class
115 class Core {
116 private:
117     int id;
118     Cache* cache;
119     std::ifstream trace_file;
120
121     // Statistics
122     int total_cycles;
123     int idle_cycles;
124     int instruction_count;
125     bool is_stalled;
126     int stall_until_cycle;
127
128 public:
129     Core(int id, Cache* cache, const std::string& trace_filename);
130     ~Core();
131
```

```
    bool executeNextInstruction(int current_cycle);
    bool hasMoreInstructions();

    // Statistics getters
    int getTotalCycles() const { return total_cycles; }
    int getIdleCycles() const { return idle_cycles; }
    int getInstructionCount() const { return instruction_count; }
    int getReadCount() const { return cache->getReadCount(); }
    int getWriteCount() const { return cache->getWriteCount(); }
    float getMissRate() const { return cache->getMissRate(); }
    int getEvictions() const { return cache->getEvictions(); }
    int getWritebacks() const { return cache->getWritebacks(); }
};

// Bus class for coherence
class Bus {
private:
    std::vector<Cache*> caches;
    int invalidations;
    int data_traffic_bytes;

public:
    Bus();

    void addCache(Cache* cache);
    void processRead(int requester_id, uint32_t addr, int& cycles_taken
    );
    void processWrite(int requester_id, uint32_t addr, int&
    cycles_taken);
    void processUpgrade(int requester_id, uint32_t addr, int&
    cycles_taken);

    // Statistics getters
    int getInvalidations() const { return invalidations; }
    int getDataTraffic() const { return data_traffic_bytes; }
    void incrementInvalidations() { invalidations++; }
    void addDataTraffic(int bytes) { data_traffic_bytes += bytes; }
};

// Main simulator class
class CacheSimulator {
private:
    std::vector<std::unique_ptr<Core>> cores;
    std::vector<std::unique_ptr<Cache>> caches;
    std::unique_ptr<Bus> bus;
    std::string app_name;
    std::string output_filename;
    int s_bits;  // Number of set index bits
    int assoc;   // Associativity
    int b_bits;  // Number of block bits

    // Random seed for tie-breaking
    int seed;

public:
    CacheSimulator(const std::string& app_name, int s, int E, int b,
                   const std::string& output_file, int random_seed = 0)
    ;
```

```
186
187     void run();
188     void outputResults();
189     int getMaxExecutionTime();
190 };
191
192 #endif // CACHE_SIMULATOR_H
```