Cache Simulator Implementation Report

Pallav Kamad 2023CS51067 Shreeraj Jambhale 2023CS50048

Abstract

This report details the implementation of a cache simulator for a four-core system using the MESI coherence protocol. The simulator models private L1 caches with write-back, write-allocate policies and LRU replacement. Key features include bus arbitration, snooping-based coherence, and detailed performance statistics. This document explains the high-level design, core classes, data structures, and critical workflows, providing insight into how memory operations are processed, synchronized, and optimized.

Contents

1 Implementation Overview

1.1 Simulator Components

- **CacheSimulator**: Manages global simulation state, bus arbitration, and trace loading.
- **L1Cache**: Represents each core's private cache, handling hits/misses, coherence actions, and LRU replacement.

1.2 Key Design Choices

- **MESI Protocol**: Ensures coherence via Modified, Exclusive, Shared, and Invalid states.
- **Central Snooping Bus**: Serializes transactions using a FIFO queue with priority for the lowest core ID.
- **Cycle-Accurate Modeling**: Tracks execution and idle cycles for each core, simulating blocking behavior during misses.
- **Statistics Collection**: Logs hits, misses, evictions, writebacks, and bus traffic for analysis.

2 Main Classes and Responsibilities

2.1 CacheSimulator Class

Roles:

- Loads memory traces from files into per-core trace_data.
- Manages the global clock (cycle), bus queue (bus_queue), and arbitration logic.
- Coordinates core execution and bus transactions.

Critical Methods:

- `runSimulation()`: Drives the simulation loop, advancing cores and resolving bus requests.
- `loadTraces()`: Parses input trace files into memory references.

2.2 L1Cache Class

Roles:

- Processes load/store requests, triggering hits, misses, or coherence operations.
- Implements LRU replacement and MESI state transitions.
- Snoops bus transactions from other cores.

Critical Methods:

- `processMemoryRequest()`: Handles hits/misses, enqueues bus requests (BUS_RD, BUS_RDX, BUS_UPGR).
- `handleBusRequest()`: Downgrades/invalidates lines during snooping.
- `completeMemoryRequest()`: Installs new cache lines after bus transactions.

3 Core Data Structures

3.1 CacheLine

```
struct CacheLine {
    bool valid;
    uint32_t tag;
    MESIState state;
    int lru_counter;
    std::vector<uint8_t> data; // Block data (size = 2^b bytes)
};
```

Tracks validity, coherence state, and LRU metadata for each cache block.

3.2 cache_sets

- A 2D vector: `std::vector<std::vector<CacheLine>> cache_sets`.
- Organized as **S sets** × **E ways** (configurable via s and E parameters).

3.3 BusRequest

```
struct BusRequest {
    int core_id;
    BusOperation operation; // BUS_RD, BUS_RDX, BUS_UPGR, FLUSH
    uint32_t address;
    int start_cycle;
    int duration;
};
```

Represents pending coherence transactions (e.g., read, invalidate, upgrade).

3.4 trace_data

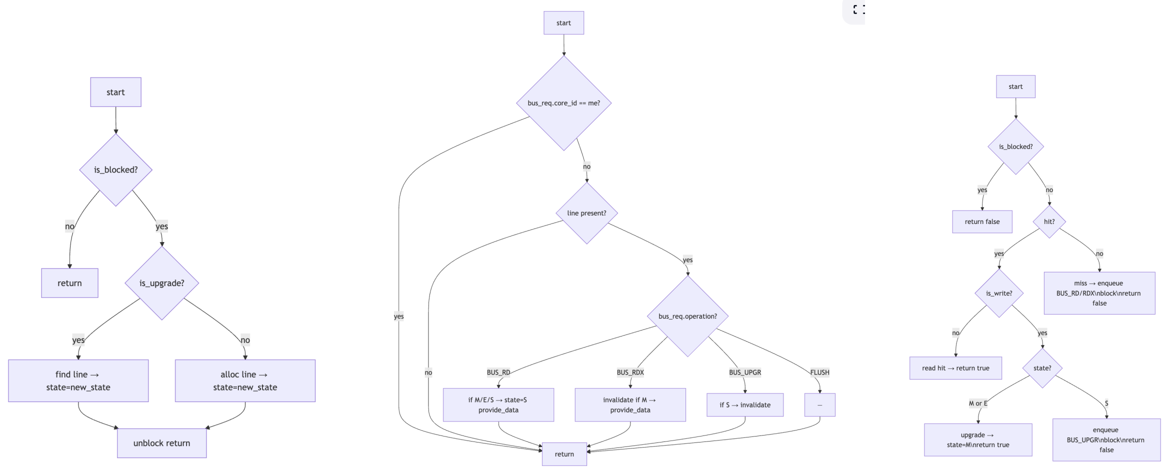
- Stores memory references for each core: `std::vector<std::vector<MemRef>> trace_data`.
- MemRef includes operation type (read/write) and address.

4 Key Function Workflows

4.1 CacheSimulator::runSimulation()

Flowchart:

1. **While** traces are not exhausted:
   - **Check Bus**: If free, dequeue the next BusRequest (lowest core ID first).
   - **Advance Cores**: For each unblocked core, process the next memory reference.
     - **Hit**: Update LRU, increment stats.
     - **Miss**: Enqueue bus requests (e.g., BUS_RD + FLUSH if evicting Modified line).
   - **Complete Bus Transactions**: Unblock cores and update cache states after duration cycles.
   - **Increment Global Clock**.



(a) completeMemoryRequest (b) handleBusRequest (c) processMemoryReq.

Figure 1: Key Cache-Coherence Diagrams

5 Performance Evaluation

5.1 Metrics Collected

- **Per-Core**:
  - Cache hits/misses, evictions, writebacks.
  - Execution cycles (active) vs. idle cycles (blocked).
- **Bus-Level**:
  - Transactions, invalidations, data traffic (bytes).

5.2 Sample Results

- **Cold Misses**: First access to a block triggers BUS_RD and installs in Exclusive state.
- **Upgrade Penalty**: A store to a Shared block issues BUS_UPGR (1 cycle), invalidating other copies.
- **Writeback Overhead**: Evicting a Modified line adds 100 cycles for FLUSH.

5.3 Optimizations Observed

- **Cache-to-Cache Transfers**: If another core holds the block, BUS_RD latency is reduced from 99 to 16 cycles (for 32-byte blocks).
- **Exclusive→Modified Transition**: No bus traffic for local writes to Exclusive blocks.

5.4 Experimental Section

- Running our simulator 10 times for the same application with the same default parameters yields the same result, as our algorithm is deterministic. This is because we resolve bus conflicts by always choosing the smallest core ID, instead of selecting one randomly. below is the plot of 10 runs

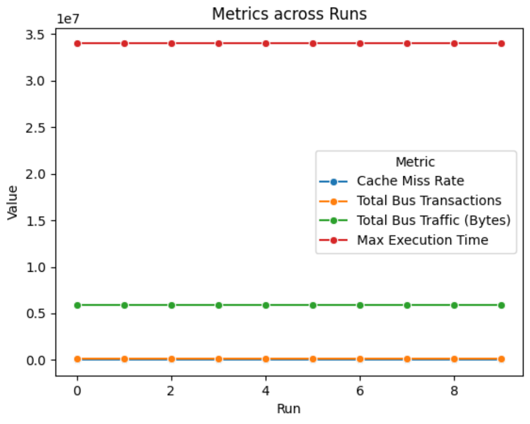


Figure 2: Output consistency across 10 simulation runs

5.5 Experimental Section 2

Running on different parameters to evaluate performance.

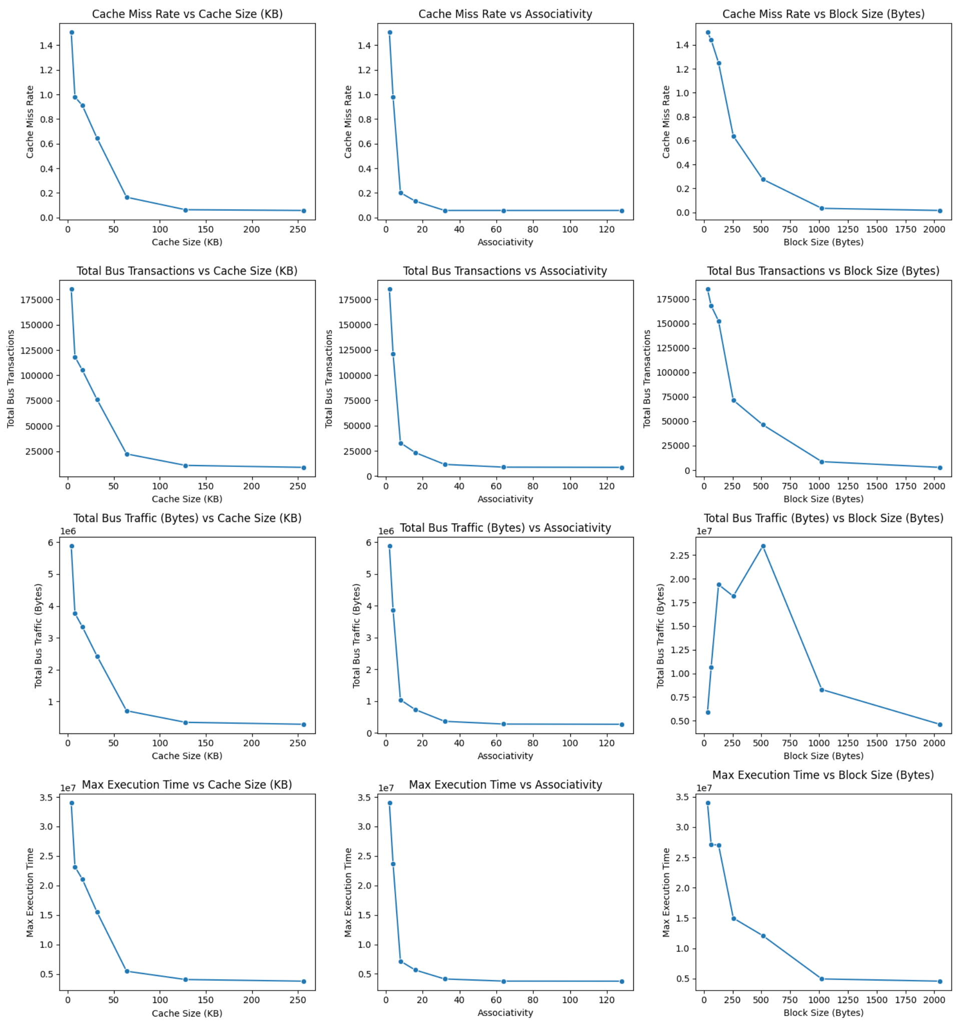


Figure 3: Performance on app1

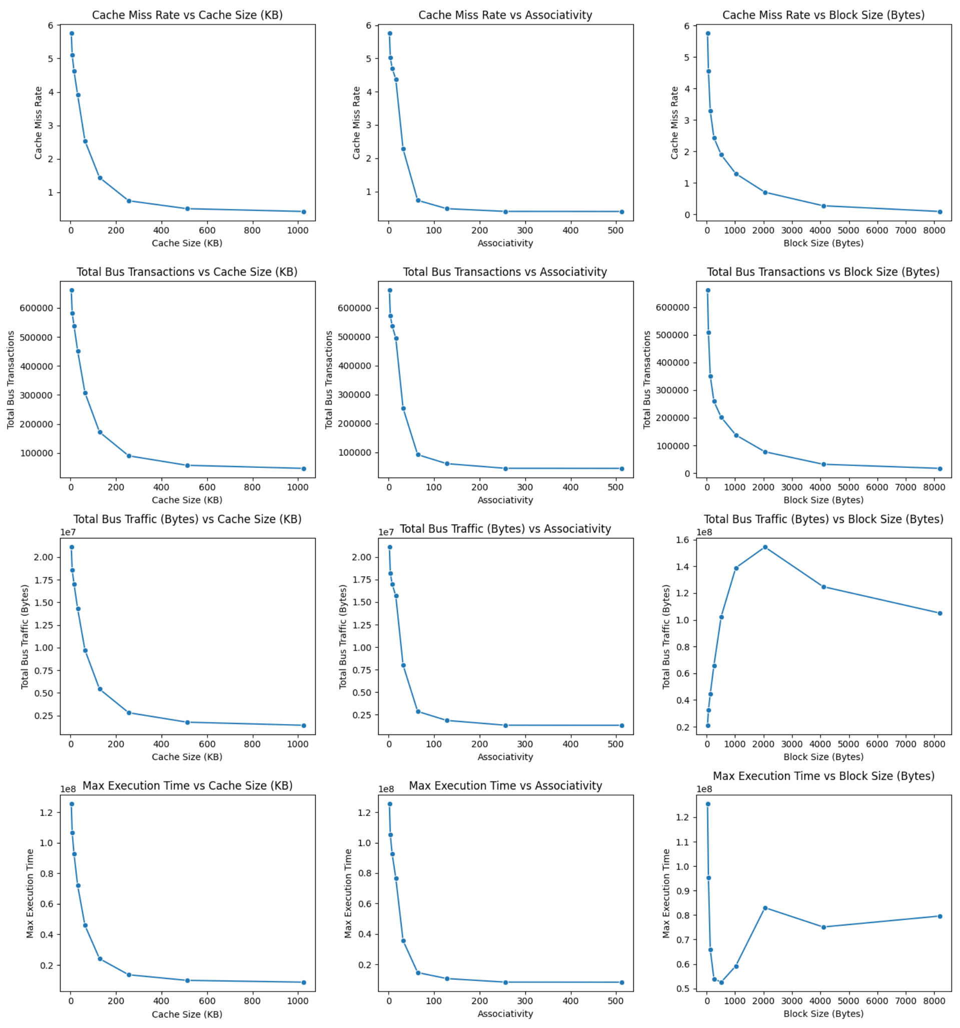


Figure 4: Performance on app2

6 Limitations and Future Work

6.1 Current Limitations

1. **Single-Level Caches**: No L2/L3 hierarchy.
2. **Blocking Design**: Cores stall during misses.
3. **Simplified Bus Model**: No pipelining or priority-based arbitration.

7 Conclusion

This simulator provides a modular, cycle-accurate model of a multicore cache system, emphasizing MESI coherence and LRU replacement. By tracking per-core and bus-level metrics, it enables analysis of spatial/temporal locality, invalidation overheads, and contention. The implementation's clarity and extensibility make it a foundation for exploring advanced features like nonblocking caches, prefetching, and heterogeneous architectures.

MESI Protocol State Transitions with Cycle Costs

Processor-Initiated Transitions

Transition	Trigger	Process	Cycles	Bus Usage
I → E	Read miss, no peer	Issue `BUS_RD`, fetch from memory, install `Exclusive`, record hit.	100+1=101	Bus busy for 100 cycles.
I → S	Read miss, peer supplies	Issue `BUS_RD`, cache-to-cache transfer, install `Shared`.	2N+1	Bus busy during transfer.
I → M	Write miss	Issue `BUS_RDX`, invalidate peers, fetch memory, install `Modified`.	100+1=101	Bus busy for 100 cycles.
S → M	Write hit on `Shared`	Issue `BUS_UPGR`, invalidate peers, upgrade to `Modified`.	1	Instant invalidation.
E → M	Write hit on `Exclusive`	Local state change to `Modified`.	1	None.
Read hit	Read in S/E/M	Return data from cache.	1	None.
Write hit	Write in M	Local update in cache.	1	None.

Table 1: Processor-Initiated MESI Transitions

Bus-Observed Transitions (Snooping)

Transition	Trigger	Process	Cycles	Core Stall
M → S	`BUS_RD` from peer	Downgrade to `Shared`, supply data cache-to-cache.	2N	No stall.
M → I	`BUS_RDX` from peer	Write back, invalidate local copy.	100	No stall.
E → S	`BUS_RD` from peer	Downgrade to `Shared`, supply data.	2N	No stall.
E → I	`BUS_RDX` from peer	Invalidate local `Exclusive`.	2N	No stall.
S → I	`BUS_RDX/BUS_UPGR`	Invalidate local `Shared`.	0	No stall.

Table 2: Bus-Observed MESI Transitions