MULTI-CORE CACHE SIMULATOR

Kathula Divya Haasini
2023CS10958

Malpani Tushar
2023CS10093

April 30, 2025

1 Introduction

This project simulates a four-core processor system with a shared communication bus and uses the MESI cache coherence protocol to maintain memory consistency across cores. Each core has its own private cache, which can be configured in terms of set index bits, associativity, and block size. The simulation takes in memory access traces and tracks cache hits, misses, and coherence-related events. The goal is to analyze how well the MESI protocol performs and how much bus traffic it generates under different workloads.

2 Flow of the Simulation

This section describes how the code for a multi-core cache simulation proceeds, focusing on the sequence of operations and function flow.

2.1 Initialization Phase

The simulation begins in the `main` function with the following setup tasks:

- **Parse Command-Line Arguments:**
  - Reads arguments trace file prefix (`-t`), cache parameters (`-s`, `-E`, `-b`), output file (`-o`).
  - These configure the simulation environment.
- **Load Trace Files:**
  - For each of the four cores, opens files named `<prefix>_proc<i>.trace`, where `i = 0` to `3`.
  - Reads memory references (read `R` or write `W` with addresses) into `deque<Ref>` objects in `refq`.
- **Initialize Data Structures:**

- `Caches`: A `vector<Cache>` named `cache` with four `Cache` objects, each initialized with parameters `s`, `E`, and `b`.
- `Bus`: A `Bus` object is instantiated to manage shared access.
- `Statistics`: A `Stats` structure is initialized per core for tracking metrics like misses and cycles.

2.2 Main Simulation Loop

The `main` function then enters a `while` loop, running until all `refq` deques are empty and no pending operations remain. Each iteration models one simulation cycle, proceeding as follows:

- **Apply Planned Changes:**
  - Processes `planned_changes`, a `vector<PlannedChange>` with scheduled state transitions or invalidations.
  - For changes due in the current cycle, updates the relevant cache line's state, tag, and usage timestamp.
- **Process Pending Allocations:**
  - Checks `pending_allocations` for allocations completing this cycle.
  - On completion, updates the victim cache line with new tag and state, and calls `Cache::touch` to update usage.
- **Process Memory References for Each Core:**
  - Iterates over all four cores:
    - **Check Stall Status:**
      - If `global_cycle < stall_until[core]`, the core is stalled and stats are updated accordingly.
    - **Process Next Reference:**
      - If not stalled and `refq[core]` is not empty, pops the front reference.
      - Computes set index and tag, then checks if it is a cache hit or miss using `Cache::find_line`.
      - **On Hit:**
      - **Read:** Push back to `planned_changes` vector.
      - **Write:**
      - If the block is in E or M state, push a state transition to M into `planned_changes`.
      - If not, and the bus is free this cycle:
      - Send a bus request to invalidate copies in other caches.
      - Increment invalidations and push update to `planned_changes`.

- If the bus is not free, stall the core until the bus becomes available.
- Increment read/write counters accordingly.
- **On Miss:**
- **Bus Check:** If the bus is occupied, stall the core until it's free.
- **Search for Block in Other Caches:**
- Checks other cores for valid copies (not I) and updates two flags:
- `found_shared`: `true` if any core has the block.
- `found_mod`: `true` if any core has it in M state.
- **Check Planned Changes:** Ensures coherence with future state transitions.
- **Decide New State and Data Transfer Cost:**
- **Write Miss (isRdX):**
- If another core has the block in M state: a writeback is required, incurring 200 cycles and invalidations.
- If other cores have shared copies (but not M): fetch from memory (101 cycles), followed by invalidation.
- New state is set to E (Exclusive).
- **Read Miss:**
- If valid copies exist in other cores, then fetch (2 cycles/word), update all to S (Shared); else, fetch from memory (101 cycles), new state is E.
- **Invalidation (if needed):** Schedules invalidation for other caches for the next cycle.
- **Victim Selection and Writeback:**
- A victim line in the requesting core's cache is selected using LRU policy. If it's valid and in M state, a writeback is triggered (additional 100 cycles).
- **Bus Occupation and Allocation:**
- Compute total transfer time (data fetch + writebacks).
- Occupy bus for that duration and create `PendingAllocation`.
- **Stall Core:** Until bus transfer and allocation complete.

- **Apply Stall Requests:**
  - Updates `stall_until[core]` for affected cores.
- **Increment Global Cycle:**
  - `global_cycle` is incremented, advancing the simulation.

2.3 Key Function Interactions

- **Cache Operations:**
  - `Cache::find_line`: Searches for a line matching the tag.
  - `Cache::choose_victim`: Chooses an invalid line or LRU line.
  - `Cache::touch`: Updates line's usage timestamp.
- **Bus Management:**
  - `Bus::free_at`: Checks if the bus is available at a cycle.
  - `Bus::occupy`: Reserves the bus until a specific cycle.
- **Statistics:**
  - Updated inline in the loop, e.g., `stats[core].misses++`.

2.4 Termination and Output

- The loop exits when all references and pending operations are processed.
- The `main` function writes statistics to the output file, summarizing performance metrics such as misses, idle cycles, and total traffic.

This flow—initialization, a cycle-by-cycle loop handling memory references and state transitions, and termination—models a coherent multi-core system using caches, MESI protocol, and a shared bus.

3 Main Classes and Data Structures

To understand the inner workings of the simulation, it is essential to examine the core classes and data structures that model the cache system and the shared bus. These components handle cache operations, manage bus access, and ensure coherence across multiple cores using the MESI protocol.

3.1 Bus Struct

The `Bus` struct models the shared communication bus used by all cores to access memory and maintain cache coherence. It has the following members and methods:

- **Members:**
  - `unsigned long long busy_until`: Indicates the cycle until which the bus is occupied.

- **Methods:**
  - `Bus()`: Constructor that initializes `busy_until` to 0.
  - `bool free_at(unsigned long long cycle)`: Checks if the bus is free at the given cycle by comparing `cycle` with `busy_until`.
  - `void occupy(unsigned long long cycle, unsigned long long duration)`: Occupies the bus starting from the given `cycle` for the specified `duration`. It updates `busy_until` to the maximum of its current value and `cycle + duration`.

This struct is crucial for managing access to the shared bus, ensuring that only one core can use the bus at a time for operations like fetching data from memory or invalidating cache lines in other cores.

3.2 Cache Struct

Each core has its own private cache, represented by the `Cache` struct. This struct manages the cache's configuration, state, and operations. Key components include:

- **Members:**
  - `int S, A, B`: Represent the number of sets (2^s), associativity (number of lines per set), and block size (2^b bytes), respectively.
  - `vector<vector<Line>> sets`: A 2D vector representing the cache sets and lines. Each set contains `A` lines.
  - `unsigned long long use_counter`: A counter used to implement the Least Recently Used (LRU) replacement policy.
- **Methods:**
  - `Cache(int s, int E, int b)`: Constructor that initializes the cache with the given parameters and resizes the `sets` vector.
  - `int find_line(unsigned long long tag, int set)`: Searches for a line with the given tag in the specified set. Returns the index if found and the line is valid and not in the Invalid (I) state; otherwise, returns -1.
  - `int choose_victim(int set)`: Selects a victim line for replacement in the specified set. Prefers invalid lines; if none, chooses the least recently used line based on `last_used`.
  - `void touch(int set, int idx)`: Updates the `last_used` timestamp of the specified line to the current `use_counter` value and increments the counter.

The `Cache` struct is essential for simulating cache behavior, including hit/miss detection, replacement policies, and state transitions under the MESI protocol.

3.3 Key Types and Enums

The simulation uses several custom types and enums defined in `types.hpp` to manage cache states and operations:

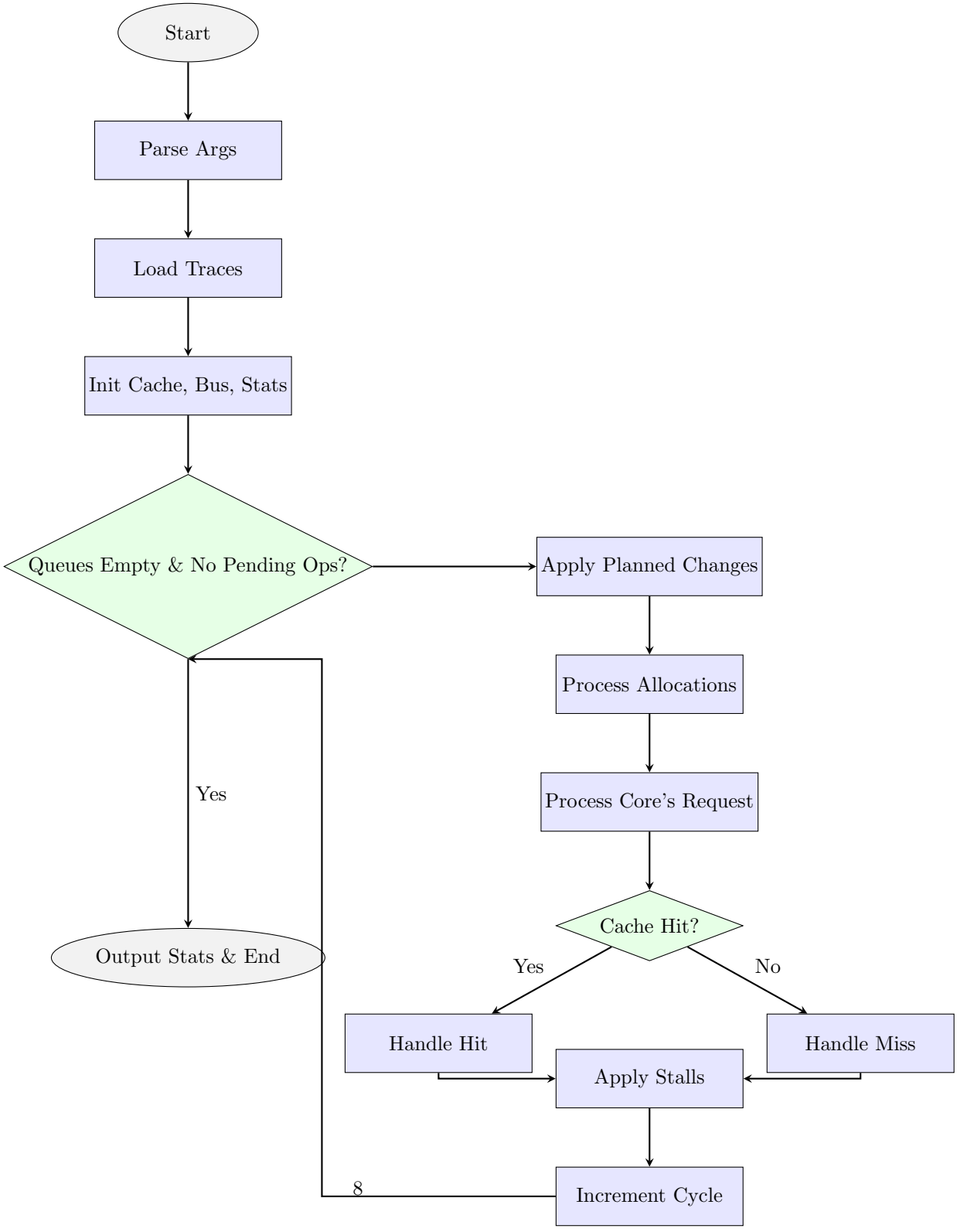
- **State Enum:**
  - Defines the possible states of a cache line under the MESI protocol: `I` (Invalid), `S` (Shared), `E` (Exclusive), and `M` (Modified).
- **Line Struct:**
  - Represents a single cache line with members:
    - `bool valid`: Indicates if the line contains valid data.
    - `State state`: The current MESI state of the line.
    - `unsigned int tag`: The tag bits of the memory address.
    - `unsigned long long last_used`: Timestamp for LRU replacement policy.
- **PendingAllocation Struct:**
  - Used to track allocations that are in progress but not yet completed. Members include:
    - `int core`: The core requesting the allocation.
    - `int set`: The set index in the cache.
    - `int victim`: The index of the victim line to be replaced.
    - `int tag`: The tag of the new block.
    - `State state`: The state the new line will be in after allocation.
    - `unsigned long long complete_cycle`: The cycle when the allocation will complete.
- **StallRequest Struct:**
  - Represents a request to stall a core until a certain cycle, typically due to bus contention or waiting for data. Members:

        * `int core`: The core to be stalled.
        * `unsigned long long until_cycle`: The cycle until which the core is stalled.

- **PlannedChange Struct:**
    - Schedules future changes to cache lines, such as state transitions or invalidations. Members include:
        * `int core`: The core whose cache will be updated.
        * `unsigned int set`: The set index.
        * `int idx`: The line index within the set.
        * `bool valid`: The new validity status.
        * `State state`: The new state.
        * `unsigned int tag`: The new tag.
        * `unsigned long long last_used`: The new last used timestamp.
        * `unsigned long longapply_cycle`: The cycle when the change should be applied.
        * `ChangeType type`: Indicates whether it's a state transition or an invalidation.

These types are critical for managing the asynchronous nature of cache operations and ensuring that changes are applied at the correct simulation cycles.

4 Flow Chart


Start
Parse Args
Load Traces
Init Cache, Bus, Stats
Queues Empty & No Pending Ops?
Yes
Output Stats & End
Apply Planned Changes
Process Allocations
Process Core's Request
Cache Hit?
Yes
No
Handle Hit
Handle Miss
Apply Stalls
Increment Cycle

5 Results Distribution

Running the cache coherence simulator with the given trace files for default parameters s = 5, E = 2, b = 5 produced the following statistics.

Table 1: App1 Performance metrics

Metric	**Core 0**	**Core 1**	**Core 2**	**Core 3**
Total Instructions	2,497,349	2,490,468	2,509,057	2,503,127
Read Instructions	1,489,888	1,485,857	1,492,629	1,493,736
Write Instructions	1,007,461	1,004,611	1,016,428	1,009,391
Execution Cycles	12,372,471	12,117,905	13,392,686	12,685,223
Idle Cycles	10,440,433	13,684,446	23,129,376	28,339,849
Cache Misses	108,768	107,961	114,056	109,240
Miss Rate	4.36%	4.33%	4.55%	4.36%
Cache Evictions	108,671	107,868	113,924	109,170
Writebacks	26,238	25,788	29,646	26,211
Bus Invalidations	45	29	60	2
Data Traffic (Bytes)	6,375,456	3,841,056	3,968,384	3,349,568

Table 2: App2 Performance metrics

Metric	**Core 0**	**Core 1**	**Core 2**	**Core 3**
Total Instructions	3,270,132	3,287,252	117,698	3,324,919
Read Instructions	2,380,720	2,388,005	74,523	2,416,052
Write Instructions	889,412	899,247	43,175	908,867
Execution Cycles	29,478,722	29,538,358	1,520,073	30,322,158
Idle Cycles	24,462,238	26,857,618	12,387,930	52,485,322
Cache Misses	234,585	233,658	10,324	235,912
Miss Rate	7.17%	7.11%	8.77%	7.10%
Cache Evictions	234,444	233,486	10,242	235,824
Writebacks	38,699	40,221	3,429	26,211
Bus Invalidations	131	80	16	4
Data Traffic (Bytes)	8,913,376	8,765,216	556,256	8,512,128

6 Graphs

6.1 Graph 1: Time vs. Number of Sets in Cache

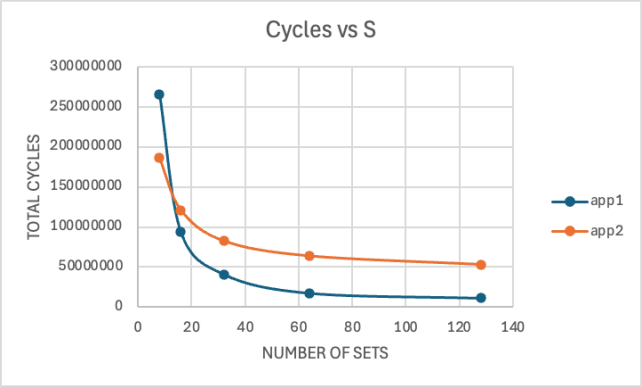


Figure 1: Total Cycles vs. Number of Sets

Observations: As the number of sets in the cache increases from 0 to 140, execution time decreases sharply (from about 260 million to 10 million cycles for app1, and from 180 million to around 50 million cycles for app2 by 64 sets) due to reduced conflicts and capacity misses. The steep initial drop reflects fewer conflicts and better data access, while the plateau after 60 sets indicates minimal further miss rate reduction. The MESI protocol effectively manages coherence, stabilizing performance with larger cache sizes.

6.2 Graph 2: Time vs. Associativity

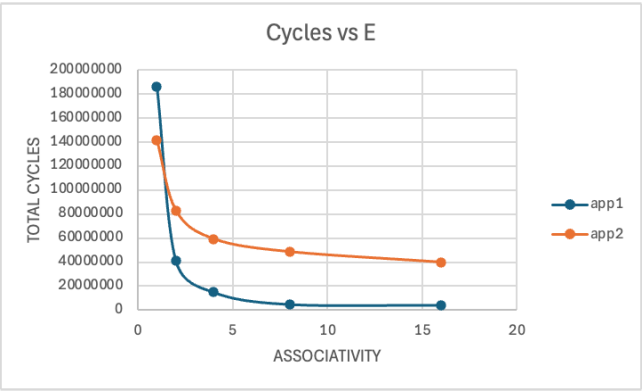


Figure 2: Total Cycles vs. Associativity

Observations: As associativity (number of blocks per set) increases from 0 to 16, execution time decreases sharply (from about 180 million to 10 million cycles for app1, and from 130 million to around 35 million cycles for app2) due to a reduced cache miss rate from fewer conflicts. The decline slows and stabilizes beyond 6 blocks, indicating the cache effectively handles the workload. This shows that higher associativity lowers the number of cache misses, thereby decreasing the execution time.

6.3 Graph 3: Total Cycles vs. Block Size

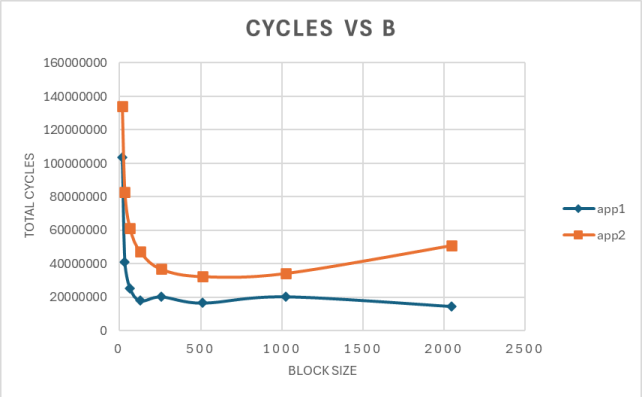


Figure 3: Total Cycles vs. Block Size

Observations: As block size increases from 0 to around 512 bytes, execution cycles initially decrease significantly (from about 130 million to 20 million cycles) due to reduced miss rates from better spatial locality—larger blocks fetch more adjacent data. However, beyond around 512 to 1024 bytes, the total cycles start to stabilize or slightly increase (reaching up to 50 million cycles by 2048 bytes for app2), because the number of blocks that can be held in the cache decreases, leading to increased conflict and capacity misses. Thus, the spatial locality benefits diminish with very large blocks, and the reduction in miss rate no longer compensates for the cache contention.

7 Assumptions

- In our processor design, the bus can handle only one request at a time. The request issued by the lower core ID is processed first, and all other cores needing the bus are stalled until the bus becomes available.
- Cores stalled due to their own requests count these cycles as execution cycles, while cores stalled due to other core requests count these cycles as idle cycles.

- When a core with data in M state supplies data to another core on a read miss, it also writes back to memory simultaneously, requiring 100 additional cycles beyond the cache-to-cache transfer time.
- Cache state transitions are implemented as planned changes that take effect in future cycles rather than immediately to maintain a clear sequence of state changes across cores.
- Invalidation messages are processed in the next cycle (1 cycle delay) after they are issued.
- The simulation considers both current cache states and pending state changes when determining coherence, to prevent race conditions where multiple cores might attempt to modify the same block simultaneously.