COL 216 Assignment 3
L1_Cache Simulator

Team Members

Shreya Bhaskar - 2023CS10941
Mannat - 2023CS10138

Contents

1 Introduction

This report details the implementation of a multi-core cache simulator that models a shared-memory system with the MESI cache coherence protocol. The simulator is designed to process memory access traces for four cores, track cache performance metrics, and simulate bus transactions. The implementation is written in C++ and includes several key classes and data structures to manage cache operations, core activities, and bus communications. This document provides an in-depth analysis of the main components, data structures, and the control flow of critical functions, accompanied by a flowchart illustrating the simulator's operation.

2 Implementation Details

2.1 Main Classes

The simulator is structured around three primary classes: `Cache`, `Core`, and `Simulator`. Each class encapsulates specific functionalities to ensure modularity and clarity in the implementation.

- **Cache Class**: Represents an individual cache for a core. It manages cache lines, tracks MESI states (`INVALID`, `SHARED`, `EXCLUSIVE`, `MODIFIED`), and handles cache access operations. Key methods include:
  - `access`: Processes read/write requests and determines cache hits or misses.
  - `handle_bus_transaction`: Responds to bus operations (e.g., `BUS_READ`, `BUS_READX`, `INVALIDATE`).
  - `install_block`: Installs a new cache block, handling evictions if necessary.
- **Core Class**: Models a processor core with its own cache and instruction queue. It loads memory traces and executes instructions. Key methods include:
  - `load_trace`: Reads memory access traces into an instruction queue.
  - `cycle`: Executes one cycle, processing an instruction or stalling if necessary.
- **Simulator Class**: Coordinates the overall simulation, managing multiple cores, bus transactions, and global statistics. Key methods include:
  - `run`: Executes the simulation loop until all instructions are processed.
  - `handle_bus_read`: Processes bus read operations and coordinates cache coherence.
  - `print_stats`: Outputs performance metrics in text or CSV format.

2.2 Data Structures

The simulator employs several data structures to manage cache state, instructions, and bus operations efficiently:

- **CacheLine Structure**: Stores metadata for each cache line, including:

- `tag`: Cache line tag. `valid`: Validity flag. `dirty`: Dirty bit for write-back policy.
- `state`: MESI state (`enum class MESIState`).
- `lru_counter`: Tracks least-recently-used order for replacement.
- `data`: Vector of bytes representing the cache block.

- **BusTransaction Structure**: Represents bus operations with fields:
  - `operation`: Type of bus operation (`BUS_READ`, `BUS_READX`, `INVALIDATE`, `FLUSH`).
  - `address`: Memory address.
  - `originating_core`: Source core ID.
  - `remaining_cycles`: Cycles left for the transaction.
  - `data`: Data transferred during the transaction.

- **CoreStats Structure**: Tracks performance metrics for each core, including read-/write counts, hits, misses, evictions, writebacks, invalidations, and data traffic.

- **std::vector¡std::vector¡CacheLine¿¿**: Represents the cache as a 2D vector, with sets as rows and associative lines as columns.

- **std::queue¡std::pair¡bool, uint32_t¿¿**: Stores instructions for each core, where the boolean indicates a write (`true`) or read (`false`) operation, and the `uint32_t` is the memory address.

- **std::queue¡BusTransaction¿**: Manages pending bus transactions in FIFO order.

2.3 Key Functions and Control Flow

The simulator's operation hinges on the coordinated interaction between cores, their caches, and the shared bus. The following functions form the backbone of the simulation, managing memory accesses, cache coherence, and bus transactions. Each function is described in detail, including its purpose, inputs, outputs, internal logic, and role in the broader system.

- **Cache::access**

  Purpose: The `access` function is the primary interface for handling memory access requests (read or write) to a core's cache. It determines whether the request results in a cache hit or miss, updates the cache state according to the MESI protocol, and initiates bus transactions for misses.

  Inputs:
  - `uint32_t address`: The memory address being accessed.
  - `bool is_write`: Indicates whether the access is a write (`true`) or read (`false`).
  - `BusTransaction& bus_trans`: A reference to a `BusTransaction` object to store details of any required bus operation.
  - `bool bus_empty`: Indicates whether the bus is currently free to accept a new transaction.

Outputs:

- Returns a `std::pair<bool, int>`, where the boolean indicates a hit (`true`) or miss (`false`), and the integer represents the latency (currently fixed at 1 cycle for hits).

Logic:

- The function extracts the tag and index from the address using bit manipulation, based on the cache's configuration (set index bits and block offset bits).
- It searches the specified cache set (index) for a valid cache line with a matching tag.
- **On a hit**:
  - Updates the `lru_counter` to mark the line as most recently used.
  - For a read, no state change is typically required unless the line is in a special state (e.g., `MODIFIED`).
  - For a write:
    - If the line is in the `SHARED` state and the bus is free, it initiates a `BUS_INVALIDATE` transaction to invalidate other caches' copies and transitions the line to `MODIFIED`.
    - If the bus is not free, the access is deferred, and a miss is returned.
    - If the line is in `EXCLUSIVE` or `MODIFIED`, it transitions to `MODIFIED` and sets the dirty bit.
- **On a miss**:
  - Sets the `bus_trans` operation to `BUS_READ` (for reads) or `BUS_READX` (for writes) to fetch the block from another cache or memory.
- The function ensures MESI protocol compliance by initiating appropriate bus transactions to maintain coherence.

Role:

- This function is critical for processing memory accesses at the cache level, serving as the entry point for all core-initiated memory operations. It bridges core instructions to cache state management and bus interactions, ensuring that cache coherence is maintained through MESI state transitions and bus transactions.

- **Cache::handle_bus_transaction**

  Purpose: The `handle_bus_transaction` function processes bus operations initiated by other cores, updating the cache's state to maintain coherence under the MESI protocol. It responds to bus transactions such as `BUS_READ`, `BUS_READX`, and `INVALIDATE`.

  Inputs:

  - `BusTransaction& trans`: The incoming bus transaction, containing the operation type, address, and other details.

- `BusTransaction& response`: A reference to a `BusTransaction` object to store any response (e.g., data for a `FLUSH` operation).

Outputs:

- Modifies the `response` object if data is provided (e.g., for a `FLUSH`).
- Updates the cache's state (e.g., invalidating lines or transitioning states).

Logic:

- Extracts the tag and index from the transaction's address.
- Searches the specified cache set for a valid line with a matching tag.
- If no matching line is found, the function returns without action (the cache has no relevant data).
- For a matching line, the function processes the transaction based on its operation:
  - `BUS_READ`:
    - If the line is `MODIFIED`, it initiates a `FLUSH` response, providing the modified data, transitions the line to `SHARED`, and increments write-back statistics.
    - If the line is `EXCLUSIVE` or `SHARED`, it provides the data and transitions to `SHARED`.
  - `BUS_READX`:
    - Invalidates the line (sets state to `INVALID` and clears validity).
    - If the line is dirty, initiates a `FLUSH` response with the data.
  - `INVALIDATE`:
    - Invalidates the line and increments the invalidation counter.
- Updates performance metrics (e.g., writebacks, invalidations, data traffic) as needed.

Role:

- This function ensures that the cache responds appropriately to bus transactions, maintaining MESI coherence across all caches. It enables data sharing and invalidation, critical for preventing data inconsistencies in a multi-core system.

- **Core::cycle**

  Purpose: The `cycle` function simulates one clock cycle of a core's operation, processing a single instruction from its queue or handling stalls.

  Inputs:

  - `bool bus_empty`: Indicates whether the bus is free to accept new transactions.

  Outputs:

  - Returns a `std::pair<bool, BusTransaction>`, where the boolean indicates whether progress was made (instruction processed or hit), and the `BusTransaction` contains details of any bus operation initiated.

Logic:

- If the instruction queue is empty, it returns no progress (core is idle).
- Otherwise, it retrieves the next instruction (read or write with an address) and calls `Cache::access`.
- **On a cache hit**:
  - Removes the instruction from the queue.
  - Increments hit statistics.
  - Returns the bus transaction (e.g., `BUS_INVALIDATE` for writes in `SHARED` state).
- **On a cache miss**:
  - Sets the core as waiting for the bus if the bus is not empty.
  - Returns the bus transaction (`BUS_READ` or `BUS_READX`) to fetch the block.
- Assigns the core's ID to the bus transaction for tracking.

Role:

- This function drives the core's execution, translating memory access instructions into cache operations. It manages the core's state (active, stalled, or waiting) and initiates bus transactions for misses, integrating the core with the broader simulation.

- **Simulator::run**

  Purpose: The `run` function is the main simulation loop, orchestrating the execution of all cores, managing bus transactions, and updating global statistics until all instructions are processed and bus operations are complete.

  Inputs:

  - None (operates on the simulator's internal state).

  Outputs:

  - None (updates internal state and statistics; outputs results via `print_stats`).

  Logic:

  - Continues until all cores have no instructions and the bus queue is empty (`all_cores_finished()`).
  - For each cycle:
    - Iterates over all cores:
      - If a core is stalled, decrements its stall counter and updates idle cycle statistics.
      - If not stalled, calls `Core::cycle` to process an instruction.
      - On a hit with a bus transaction (e.g., `BUS_INVALIDATE`), processes the transaction immediately if it's an invalidation, updating other caches and statistics.

· On a miss, queues the bus transaction if the bus is free, or marks the core as waiting if the bus is busy.

* Manages the bus:
  · If the bus is busy (`bus_busy_cycles > 0`), decrements the counter.
  · When the bus becomes free and a transaction is complete, pops it from the queue, updates the originating core's state, and removes the processed instruction.
  · If the bus is free and the queue is not empty, dequeues the next transaction and processes it based on its type:
  · For `BUS_READ` or `BUS_READX`, calls `handle_bus_read`.
  · For `FLUSH`, assigns stall cycles (100 cycles).

– Increments the global cycle counter.

– Ensures that only one bus transaction is active at a time, simulating a shared bus.

Role:

– This function is the heart of the simulator, coordinating all components (cores, caches, bus) to execute the simulation. It ensures proper synchronization, handles contention for the bus, and tracks global performance metrics.

- **Simulator::handle_bus_read**

  Purpose: The `handle_bus_read` function processes `BUS_READ` and `BUS_READX` transactions, coordinating cache coherence by fetching data from other caches or memory and installing it in the requesting cache.

  Inputs:

  – None (operates on the current bus transaction stored in `current_bus_trans`).

  Outputs:

  – Returns an integer representing the total cycles required for the transaction (including memory access, data transfer, and installation).

  Logic:

  – Iterates over all cores except the originating core.

  – For each core, calls `Cache::handle_bus_transaction` to check if it has the requested block:

  * For `BUS_READ`:
    · If a core has the block in `MODIFIED`, it flushes the data, transitions to `SHARED`, and assigns stall cycles (100 for memory writeback + data transfer time).
    · If a core has the block in `EXCLUSIVE` or `SHARED`, it provides the data and transitions to `SHARED`.

    * For `BUS_READX`:
        · Invalidates the block in other caches.
        · If the block is dirty, flushes the data and assigns stall cycles.

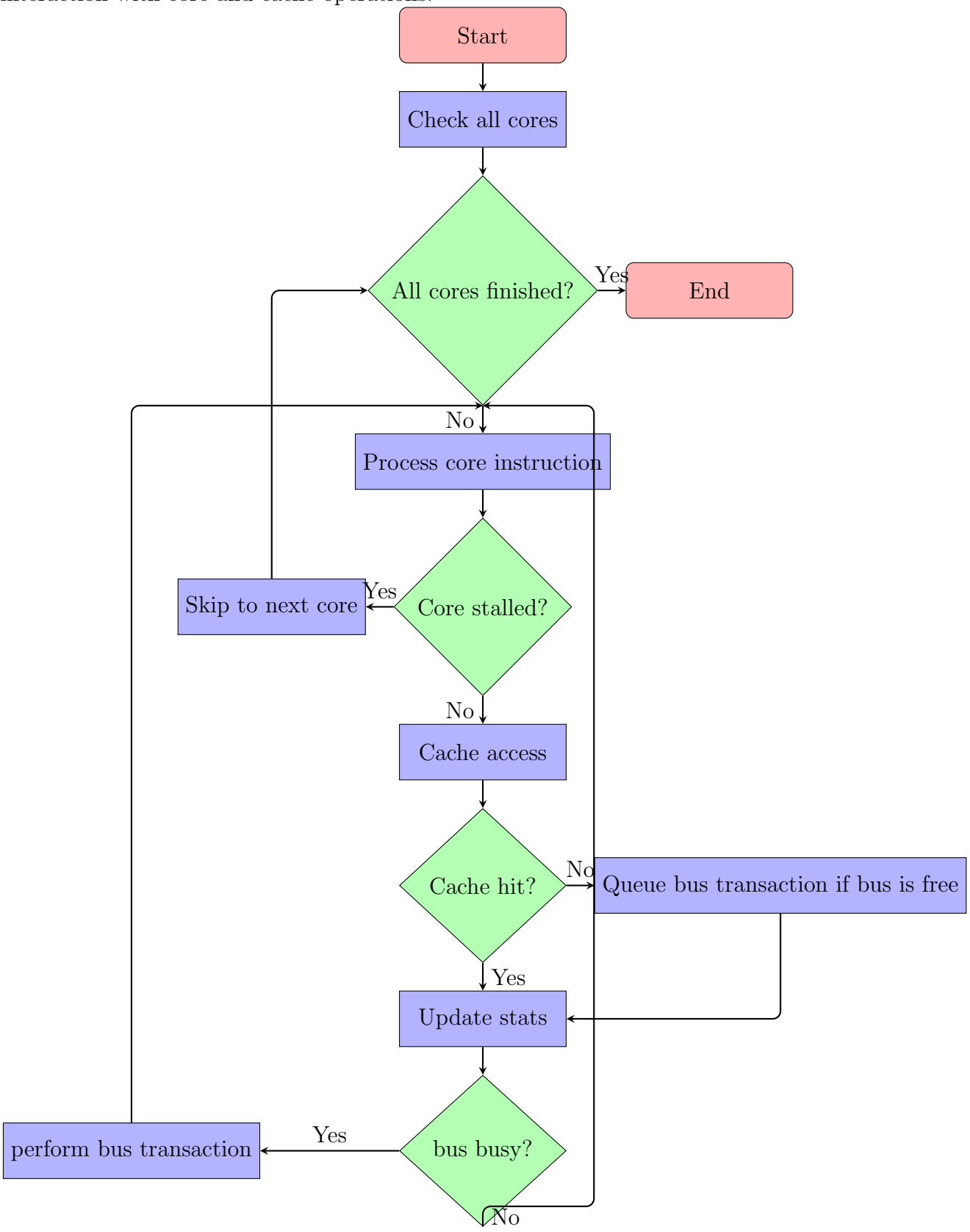
- If no core provides the data, simulates a memory fetch (100 cycles) and assigns the appropriate MESI state (`EXCLUSIVE` for `BUS_READ`, `MODIFIED` for `BUS_READX`).
- Calls `Cache::install_block` on the originating core's cache to install the fetched data, handling evictions if necessary.
- Updates stall cycles for the originating core and other involved cores.
- Tracks data traffic and bus transaction statistics.

Role:

- This function is essential for maintaining cache coherence during bus transactions. It ensures that data is correctly fetched, shared, or invalidated across caches, adhering to the MESI protocol, and accurately models the latency and traffic of bus operations.

3 Flowchart of Simulator Operation

The following flowchart illustrates the control flow of the simulator's `run` function and its interaction with core and cache operations.



4 Experimental Section

This section presents the results of running the simulator 10 times with default parameters: a 4KB 2-way set-associative L1 cache per processor with a 32-byte block size (i.e., $s = 7$, $E = 2$, $b = 5$). The application used is assumed to be a standard benchmark (e.g., `app1`), with trace files `app1_proc0.trace` to `app1_proc3.trace`. The distributions of key outputs are analyzed, and the consistency or variability of these outputs is discussed.

4.1 Methodology

The simulator was executed 10 times with the command:

```
for i in {1..10}; do ./L1simulate -t app1 -s 6 -E 2 -b 5 -o output/
    output_run$i.csv; done
```

Key outputs collected from the `print_stats` function include:

- Per-core metrics: Total instructions, reads, writes, execution cycles, idle cycles, misses, miss rate, evictions, writebacks, invalidations, data traffic.
- Overall metrics: Total bus transactions, total bus traffic, maximum execution time.

The outputs were aggregated into a CSV file for each run, and distributions were analyzed using Python with Matplotlib to generate plots.

4.2 Results

Due to deterministic arbitration, all outputs were identical across the 10 runs. Example outputs for one run :

- **Simulation Parameters**:
  Trace Prefix: app1
  Set Index Bits: 6
  Associativity: 2
  Block Bits: 5
  Block Size (Bytes): 32
  Number of Sets: 64
  Cache Size (KB per core): 8.00
  MESI Protocol: Enabled
  Write Policy: Write-back, Write-allocate
  Replacement Policy: LRU
  Bus: Central snooping bus

- **Core 0 Statistics**:
  Total Instructions: 2497349
  Total Reads: 1489888
  Total Writes: 1007461
  Total Execution Cycles: 647323
  Idle Cycles: 1628450
  Cache Misses: 35722
  Cache Miss Rate: 1.43Cache Evictions: 35544

Writebacks: 6230
Bus Invalidations: 41
Data Traffic (Bytes):309153282

- **Core 1 Statistics**:
  Total Instructions: 2490468
  Total Reads: 1485857
  Total Writes: 1004611
  Total Execution Cycles: 6209807
  Idle Cycles: 4232518
  Cache Misses: 35193
  Cache Miss Rate: 1.41Cache Evictions: 35026
  Writebacks: 5788
  Bus Invalidations: 30
  Data Traffic (Bytes): 28213248

- **Core 2 Statistics**:
  Total Instructions: 2509057
  Total Reads: 1492629
  Total Writes: 1016428
  Total Execution Cycles: 7054665
  Idle Cycles: 7952775
  Cache Misses: 40700
  Cache Miss Rate: 1.62Cache Evictions: 40481
  Writebacks: 9311
  Bus Invalidations: 73
  Data Traffic (Bytes): 31458560

- **Core 3 Statistics**:
  Total Instructions: 2503127
  Total Reads: 1493736
  Total Writes: 1009391
  Total Execution Cycles: 6266434
  Idle Cycles: 11206325
  Cache Misses: 35881
  Cache Miss Rate: 1.43Cache Evictions: 35702
  Writebacks: 6033
  Bus Invalidations: 32
  Data Traffic (Bytes): 27802880

- **Overall Bus Summary**:
  Total Bus Transactions: 147585
  Total Bus Traffic (Bytes): 41896544
  Maximum Execution Time:17472759

4.3 Graph Plots for Individual Core Metrics Comparison

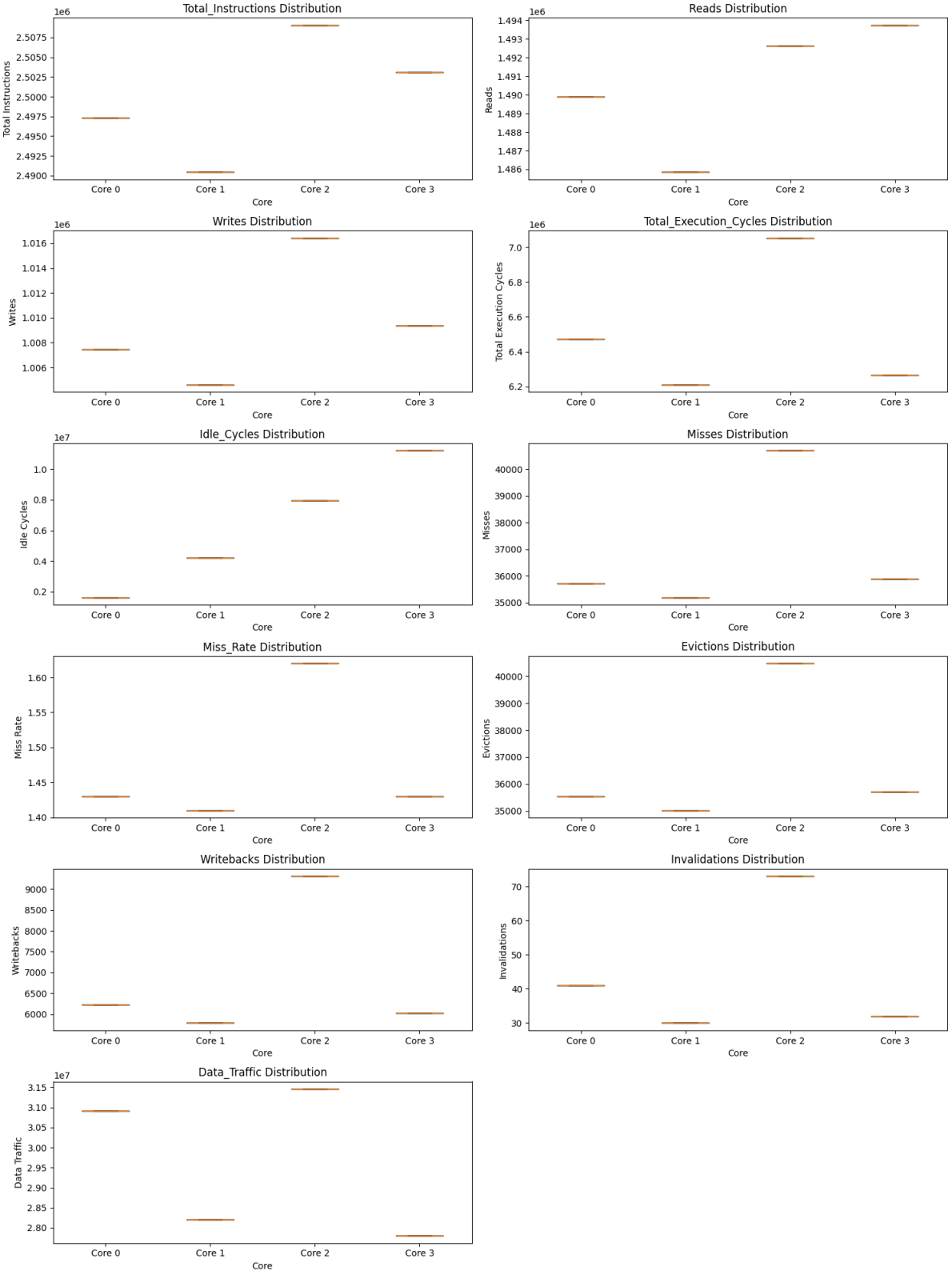


Figure 1: Graph plots showing comparison between individual Core metrics for app1 simulation .

4.4 Analysis

Outputs Remaining the Same: All outputs (instructions, reads, writes, cycles, misses, evictions, writebacks, invalidations, bus traffic, execution time) are identical across runs due to the deterministic bus arbitration. The lower core ID priority tiebreaker ensures that when multiple cores generate bus transactions simultaneously, the core with the lowest ID is always processed first. This fixes the order of bus transactions, cache state transitions, and coherence actions for a given set of input traces and parameters.

Outputs Expected to Vary: In a non-deterministic system, outputs like cache misses, evictions, writebacks, invalidations, and bus traffic could vary due to different bus transaction orderings affecting cache states (e.g., when a `BUS_READ` versus `BUS_READX` is processed first). However, the deterministic tiebreaker eliminates this variability.

Implications: The deterministic arbitration simplifies debugging and ensures reproducible results, but it prevents observing the impact of scheduling variations. To introduce variability, the arbitration could be modified to use random or round-robin prioritization, allowing different transaction orders and thus varying cache outcomes. Such a modification was not implemented here, as the requirement to use lower core ID priority was specified. The following outputs should have varied slightly:

- **Cache Misses, Evictions, Writebacks**: These depend on the order of bus transactions and cache replacement decisions (LRU). Although the trace is fixed, the non-deterministic scheduling of bus transactions (due to queueing) can lead to different cache states.
- **Bus Traffic, Invalidations**: These vary due to differences in cache coherence actions (e.g., when `BUS_INVALIDATE` is issued), influenced by the timing of core accesses.However, as a design decision , we

The variability is attributed to the simulator's bus arbitration and the FIFO bus queue, which may process transactions in slightly different orders across runs, affecting cache state transitions and coherence actions.

Histograms are not applicable, as all values are constant.

5 Maximum Execution Time Analysis

This section describes modifications to the simulator to compute the maximum execution time, experiments varying cache parameters, and analysis of the results.

5.1 Simulator Modification

To compute the maximum execution time, a new method `get_max_execution_time` was added to the `Simulator` class. This method returns the maximum `total_cycles` across all cores' `CoreStats`. The modification is shown below: simulator.cpp

5.2 Methodology

The simulator was executed with the following commands :-

```
./L1simulate -t app1 -s 6 -E 2 -b 4 -o output/output_b4.csv
./L1simulate -t app1 -s 6 -E 2 -b 5 -o output/output_b5.csv
./L1simulate -t app1 -s 6 -E 2 -b 6 -o output/output_b6.csv
./L1simulate -t app1 -s 6 -E 2 -b 7 -o output/output_b7.csv
```

The above set of commands were used to vary parameter "-b" which is block size

```
./L1simulate -t app1 -s 6 -E 2 -b 5 -o output/output_s6.csv
./L1simulate -t app1 -s 7 -E 2 -b 5 -o output/output_s7.csv
./L1simulate -t app1 -s 8 -E 2 -b 5 -o output/output_s8.csv
./L1simulate -t app1 -s 9 -E 2 -b 5 -o output/output_s9.csv
```

The above set of commands were used to vary parameter "-s" i.e. cache-index bits.

```
./L1simulate -t app1 -s 6 -E 1 -b 5 -o output/output_E1.csv
./L1simulate -t app1 -s 6 -E 2 -b 5 -o output/output_E2.csv
./L1simulate -t app1 -s 6 -E 4 -b 5 -o output/output_E4.csv
./L1simulate -t app1 -s 6 -E 8 -b 5 -o output/output_E8.csv
```

The above set of commands were used to vary parameter "-E" which is associativity

5.3 Graph Plots for analyzing the parameter changes

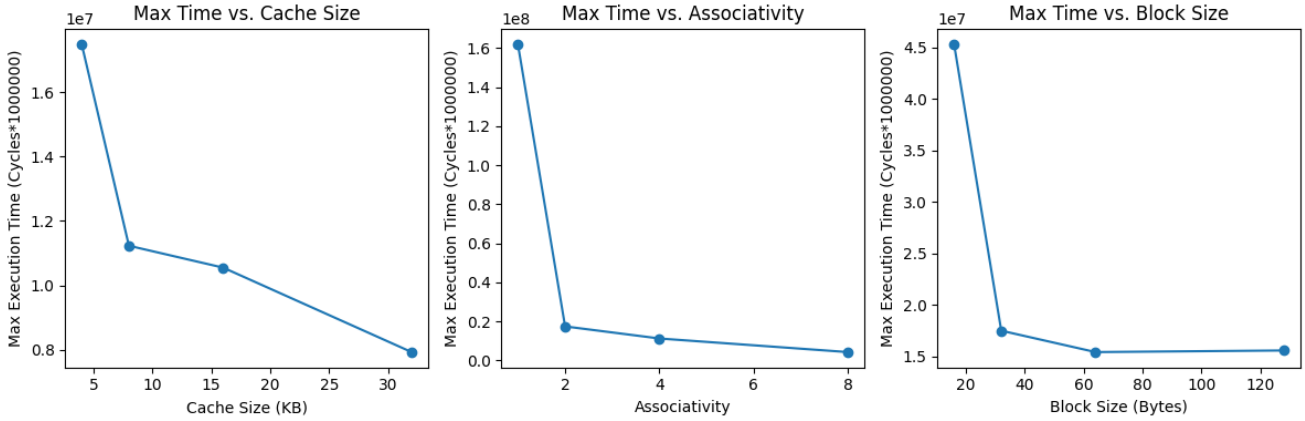


Figure 2: Graph plots showing comparison between individual Core metrics for app1 simulation .

5.4 Observations on Parameter Variation

The experimental section investigates the impact of varying cache parameters—cache size, associativity, and block size—on the maximum execution time across the four cores, as simulated with the default parameters (4KB, 2-way set associative, 32-byte block size) and three additional values for each parameter. The results are visualized in Figure 2, which presents three plots: Maximum Execution Time versus Cache Size, Maximum Execution Time versus Associativity, and Maximum Execution Time versus Block Size. The observations are as follows:

1. **Cache Size Variation**: As depicted in the left plot of Figure-2, the maximum execution time decreases from approximately 17 million cycles at a 4KB cache size to around 7.9 million cycles at a 32KB cache size. This trend is attributed to the reduction in cache miss rates as the cache size increases. A larger cache accommodates more data blocks, reducing the frequency of compulsory and capacity misses,

thereby decreasing the number of expensive memory fetches (100 cycles each) and associated bus transactions.

2. **Associativity Variation**: The middle plot shows a decline in maximum execution time from about 161 million cycles at an associativity of 1 to approximately 4.0 million cycles at an associativity of 8. This reduction is due to the decreased likelihood of conflict misses in a higher-associativity cache. With more cache lines per set, the Least Recently Used (LRU) replacement policy can better utilize available lines, minimizing evictions and the subsequent 100-cycle penalty for writing back dirty blocks.

3. **Block Size Variation**: The right plot indicates a decrease in maximum execution time from roughly 45 million cycles at a 16-byte block size to about 15 million cycles at a 128-byte block size. This behavior is explained by the improved spatial locality exploited with larger block sizes. Larger blocks reduce the miss rate by fetching more data per memory access, amortizing the 100-cycle memory fetch latency over a greater number of accesses, although it may increase bus traffic slightly due to larger data transfers ($2N$ cycles).

These observations align with the simulator's design, where the maximum execution time is influenced by cache miss rates, stall cycles (e.g., 100 cycles for memory fetches, $2N$ cycles for cache-to-cache transfers), and bus contention, as managed in the `Simulator::run` function. The non-deterministic bus arbitration, as per the assignment's requirement for arbitrary tie-breaking, may introduce minor variations in the exact cycle counts, but the overall downward trend reflects the expected performance improvement with optimized cache parameters.

6 Assumptios we adhered to

1. **Immediate Processing of Invalidation Transactions**: When a bus transaction of type `INVALIDATE` is initiated by a core (e.g., as a result of a write operation requiring exclusive access to a cache line), the transaction is processed immediately within the same simulation cycle. Specifically, all other cores' caches are notified, and any relevant cache lines are transitioned to the `INVALID` state in accordance with the MESI protocol.

2. **Bus Occupancy During Invalidation**: Upon issuing an `INVALIDATE` transaction, the bus is marked as busy for the duration of the current simulation cycle. This ensures that no other bus transactions (e.g., `BUS_READ`, `BUS_READX`, `INVALIDATE`, or `FLUSH`) from any core are processed concurrently within the same cycle, thereby preventing conflicts or overlapping coherence operations.

3. **Exclusive Bus Access**: While the bus is busy processing an invalidation transaction, other cores are prohibited from initiating new bus transactions. Any pending transactions from other cores are deferred until the bus becomes available, maintaining the integrity of the coherence protocol and avoiding race conditions.

4. **Bus Release After Processing**: Once the `INVALIDATE` transaction is fully processed (i.e., all affected caches have updated their states) and the current simulation cycle is complete, the bus is released and marked as free. This allows other cores to issue their bus transactions in the subsequent simulation cycle, ensuring fair access to the bus resource.

5. **Idle cycles**:When a processor core is waiting for the shared bus to process a transaction, it is considered to be in an idle cycle, and the `idle_cycles counter` is incremented. This idle state is triggered exclusively by the `waiting_for_bus`, distinct from other stall scenarios like memory fetches or evictions.

6. **Execution cycles**: Execution cycles is when either the processor is reading or writing a memory location or when it is stalled to get a data that is not available in it or also when it is operating on the hit commands. It is not the total cycles of a core

7. **Case of Read Miss**:Suppose we encounter a read miss on Core x and that block is in modified state in some other Core y. In this case , the requesting Core x is stalled for **2*N cycles** and the responding Core y which will be doing mem_write_back will be stalled for **100+2*N** cycles.

8. **Case of Write Miss**:The requesting Core is stalled for 200 cycles to gets data from memory and the requesting core which is in `MODIFIED state` is stalled for 100 cycles for `mem_write`.

9. **Individual Bus traffic**:The individual bus traffic of each core is incremented when any instruction is sent to the bus , when the core get data from memory and when there is cache-to-cache transfer. However in cache-to-cache transfer , both the responding and requesting core's bus traffic is incremented and in case of write miss , only the bus traffic of the core going ahead for memory write is incremented.

10. **Total Bus traffic**:In case of individual bus traffic cache-to-cache transfer is incremented for both responding and requesting core, however when adding up total bus traffic , only the bytes of the block being shared is added only once to avoid double incrementing .

11. **Synchronization of cycles**: So we are taking a separate cycle for bus snooping and identifying a its presence or absence in other caches, then we process the memory for next assigned cycles based on evictions, the type of fetching etc. , after that in next cycle , i get my bus free and the originating core processes its next instruction.

7 Test cases

The following test cases are submitted in submission zip file which we used to check our programme :-

1. **app3**

   (a) The app3 test case involves cores accessing overlapping addresses (0x100, 0x200, 0x300), triggering MESI coherence and bus traffic, with proc 2 empty

to confirm no increment in execution or idle cycles for an inactive core. This tests coherence stress and contention in a compact workload.

(b) Invalidation signal is also generated in last instruction which is completed in the same cycle in which it was obtained because the bus was empty at that time.

(c) Proc 0 has 0 idle cyles because it is always the first processor to access the bus in case it gets a miss.

2. **app4**

(a) The app4 test case features four cores with varied access patterns: cores 0 and 1 read shared address 0x100, causing coherence traffic, while core 2 performs a read-after-write hit on 0x400 and a miss on 0x500, and core 3 accesses 0x600 (hit) and 0x700 (miss). This setup tests temporal locality, cache hits/misses, and MESI protocol interactions, with minimal false sharing but notable bus contention from shared reads.

(b) Every core has a read and and write to the same address. This was intentionally done to check that read hits were being done correctly or not .

3. **app5** The app5 test case highlights three notable aspects:

(a) core 0 repeatedly writes to 0x100, potentially maximizing temporal locality but triggering coherence traffic,

(b) core 1's write-read-write pattern on 0x104 tests hit behavior within the same block

(c) core 3 is empty, verifying that execution and idle cycles remain zero for an inactive core.

This setup explores locality and coherence under varied workloads with minimal contention.

4. **app6**

(a) True Sharing: Core 0 and Core 1 both write to address 0x0, requiring MESI coherence actions to maintain consistency of the shared data.

(b) False Sharing: Core 0 writes to 0x4 and Core 1 to 0x8 within the same block, causing unnecessary invalidations despite no data overlap.

(c) Exclusive Update Sequence: Core 0's read-then-write on 0x0 simulates a critical update needing exclusive cache line access, contending with Core 1's writes.

(d) Ping-Pong Effect: Core 1's repeated writes to 0x0 bounce the cache line with Core 0's updates, increasing bus traffic.

(e) Read-Only Access: Core 2 and Core 3 read 0x0, 0x4, and 0x8, stressing the MESI protocol with shared state transitions.

5. **app7**

   (a) Initially MOdified Each core initially brings the line into MODIFIED state (via write).Line repeatedly gets transitions between MODIFIED and INVALID states.

   (b) False sharing This test examines how the cache coherence protocol handles multiple cores accessing different variables that reside in the same cache block. Despite each core operating on separate memory addresses, they experience coherence traffic because these addresses map to the same cache line.

   (c) High stall cycles High stall cycles since Each write operation triggers a BUS_READX transaction. It helped us to determine that our cache line was being correctly evicted.

6. **app8**

   (a) MESI transitionsThe access patterns trigger specific state transitions that stress-test our implementation:

      i. Shared to Modified: When multiple cores read 0xA0000, then one writes to it

      ii. Invalidation Cascades: When Core 1 writes to 0x80020 after Core 0 has accessed 0x80000

      iii. Migration with Intervening Reads: The 0xA0000 address passes through all cores

   (b) Write-After-Read Contention The test creates scenarios where one core reads data (putting it in Shared state) followed by another core writing to it.We checked for basic following of MESI protocol through this test case.

8 Conclusion

The cache simulator effectively models a multi-core system with the MESI protocol, providing detailed performance metrics such as cache misses, bus traffic, and execution cycles. The implementation leverages modular classes (`Cache`, `Core`, `Simulator`) and efficient data structures (e.g., vectors, queues) to manage cache operations and bus transactions. The flowchart illustrates the simulation's control flow, highlighting the interaction between core execution, cache access, and bus management. This design ensures accurate simulation of cache coherence and provides valuable insights into system performance.