COL216 Assigment - 3

Ponnada Rajasree (2023CS10015) — Amagoth Saisrishanth (2023CS50696)

April 30, 2025

1 Assumptions

- **Bus Arbitration Implementation**: Implemented a deterministic **Round-Robin** arbitration scheme starting from Core 0, instead of true random arbitration, to ensure simulation reproducibility.
- **Bus Transaction Atomicity**: Once a transaction wins arbitration, it occupies the bus for its entire duration (1, 2N, or 100 cycles) without interruption. Snooping occurs concurrently.
- **BusUpgr Latency**: Assumed a minimal bus latency of **1 cycle** for the `BusUpgr` signal.
- **Data Source on BusRdX Hitting E State**: Data is assumed to **come from memory** (with 100 cycle latency) per Slide 22, despite possible C2C optimizations.
- **Definition of "Idle Cycles"**: Cycles where either the `core_stalled_on_cache` or `needs_completion_cycle` flag is true, indicating waiting for cache response or completing a miss.
- **Definition of "Total Execution Cycles" (Per Core)**: Defined as the `global_cycle` count when the **entire simulation** finishes (all cores complete traces and pending cache operations).
- **Address Decomposition**: Used low-order bits for block offset, followed by set index bits, with remaining bits as tag. Handled the case `s=0` for fully associative caches.
- **Word Size**: Consistently assumed a **4-byte word size** for calculations, including computing N for C2C latency ($N = \text{BlockSize}/4$).
- **Additional 100 Cycles**: Interpreted memory/WB latency ("100 cycles") as the time the transaction **occupies the bus**, affecting core stall time accordingly.
- **Integer Sizes**: Used `uint32_t` for addresses and `uint64_t` for cycle counts and statistic counters to prevent overflow.

2 Implementation Overview

2.1 Main Classes and Data Structures

The project consists of the following primary modules:

- **Bus**: Handles arbitration between cores for memory access, snooping, and bus transactions (read, writeback, upgrade).
- **Cache**: Implements a per-core private L1 cache with MESI coherence protocol, LRU replacement, and handles cache misses.
- **Core**: Simulates instruction execution, interacts with its private cache, and tracks stalls due to cache misses.
- **Stats**: Collects detailed statistics including instructions, stalls, cache misses, evictions, writebacks, and bus traffic.

- **Simulator**: Coordinates global clock cycles, runs cores and bus, checks simulation completion.

Key data structures include: Here's how different parts of the simulation are modeled using C++ structures and classes in the provided code:

Simulation Control

- **Simulator (Class)**: Orchestrates the entire simulation. Owns the Bus, Caches, and Cores. Manages the global cycle clock and runs the main simulation loop (`run()`). Determines overall completion.

Processor Core

- **Core (Class)**: Represents a single processor core.
  - `FILE* trace_file_ptr`: Handles reading instructions from the core's trace file efficiently using C I/O.
  - `Cache* cache`: Pointer to the specific L1 Cache associated with this core.
  - `Stats* stats`: Pointer to the shared statistics object to record core-specific and global stats.
  - `bool core_stalled_on_cache`: Flag indicating if the core is paused waiting for a cache miss to resolve.
  - `MemAccess current_access`: Struct holding the type (Read/Write) and address of the instruction currently being processed or waited upon.

L1 Data Cache

- **Cache (Class)**: Models a single L1 cache.
  - `std::vector<CacheSet> sets`: Holds all the cache sets (size = S = 2^s).
  - `Bus* bus`: Pointer to the shared bus for communication.
  - `Stats* stats`: Pointer for recording cache-related statistics (hits, misses, evictions, etc.).
  - `bool stalled`: Internal flag (can differ from Core's stall flag), possibly indicating the cache itself is busy processing a miss sequence.
  - `std::map<addr_t, PendingRequest> pending_requests`: Tracks ongoing cache misses. The key is the block address being fetched/upgraded. `PendingRequest` (struct within `Cache`) stores details like the original operation (R/W) and the cache way allocated for the incoming data.

Cache Set

- **CacheSet (Class)**: Represents a single set within the cache.
  - `std::vector<CacheLine> lines`: Holds the cache lines belonging to this set (size = E, associativity). Manages lookup and replacement within this set.

Cache Line

- **CacheLine (Struct)**: The fundamental unit of storage in the cache.
  - `MESIState state`: Stores the coherence state (Modified, Exclusive, Shared, Invalid) using an `enum class`.
  - `addr_t tag`: Stores the tag portion of the memory address associated with the data in this line.
  - `cycle_t lastUsedCycle`: Timestamp used for the LRU replacement policy.

System Bus

- **Bus (Class)**: Models the shared communication pathway.
  - `std::vector<std::queue<BusRequest>> requests_per_core`: Queues (one per core) holding pending bus transaction requests (`BusRd`, `BusRdX`, `BusUpgr`, `Writeback`).
  - `bool busy`: Flag indicating if a transaction is currently using the bus.
  - `cycle_t transaction_end_cycle`: Timestamp when the current bus transaction will complete.
  - `BusRequest current_transaction`: Struct holding details of the transaction currently occupying the bus.
  - `std::vector<Cache*> caches`: Pointers to all caches in the system for broadcasting snoop requests.
  - `int arbitration_pointer`: Used for implementing round-robin arbitration among cores requesting the bus.

Memory Access

- **MemAccess (Struct)**: Simple structure holding the operation type (`Operation` enum: `READ`/`WRITE`) and the `addr_t` address, representing a single instruction from the trace file.

Statistics Collection

- **Stats (Class)**: Centralized collector for all simulation statistics (per-core reads/writes, cycles, stalls, misses, evictions, writebacks, and bus-level traffic/invalidations). Uses various `std::vector<uint64_t>` and `uint64_t` members.

Coherence States

- **MESIState (Enum Class)**: Defines the four possible states for a cache line.

Key Data Structures

- `std::vector` containers for tracking per-core statistics.
- `std::queue` per core in `Bus` for handling pending bus requests.
- MESI state machines implemented via `enum class` inside cache lines.

Statistics Summary

- **Instructions (R/W):** Incremented via `Stats::recordAccess`, called from `Cache::access` using the operation type.
- **Total Core Cycles:** Recorded at end of simulation via `Simulator::run()` using the `global_cycle` when `checkCompletion()` returns true.
- **Idle Cycles:** Incremented using `Stats::incrementStallCycles` in `Core::tick()` if the core is stalled (`core_stalled_on_cache` is true or `needs_completion_cycle` just became true).
- **Cache Misses:** Incremented via `Stats::recordMiss` in `Cache::access` on any miss (e.g., `I->M/S/E`, `S->M`).
- **Cache Accesses:** Recorded via `Stats::recordAccess` every time `Cache::access` is invoked.
- **Cache Miss Rate:** Computed as `cache_misses / cache_accesses` at the end in `Stats::printFinalStats`.

- **Cache Evictions:** Incremented via `Stats::recordEviction` in `Cache::allocateBlock` when a valid line is evicted (i.e., `findInvalidLine()` fails and LRU chooses a valid line).
- **Writebacks:** Incremented in `Stats::recordWriteback`, called either during `Cache::allocateBlock` (victim state `M`) or during snoop handling if a line in state `M` is downgraded.
- **Max Execution Cycles:** Computed in `Stats::printFinalStats` using the max of the `total_cycles` vector (or just the final `global_cycle`).

2.2 Flowchart of Major Functions

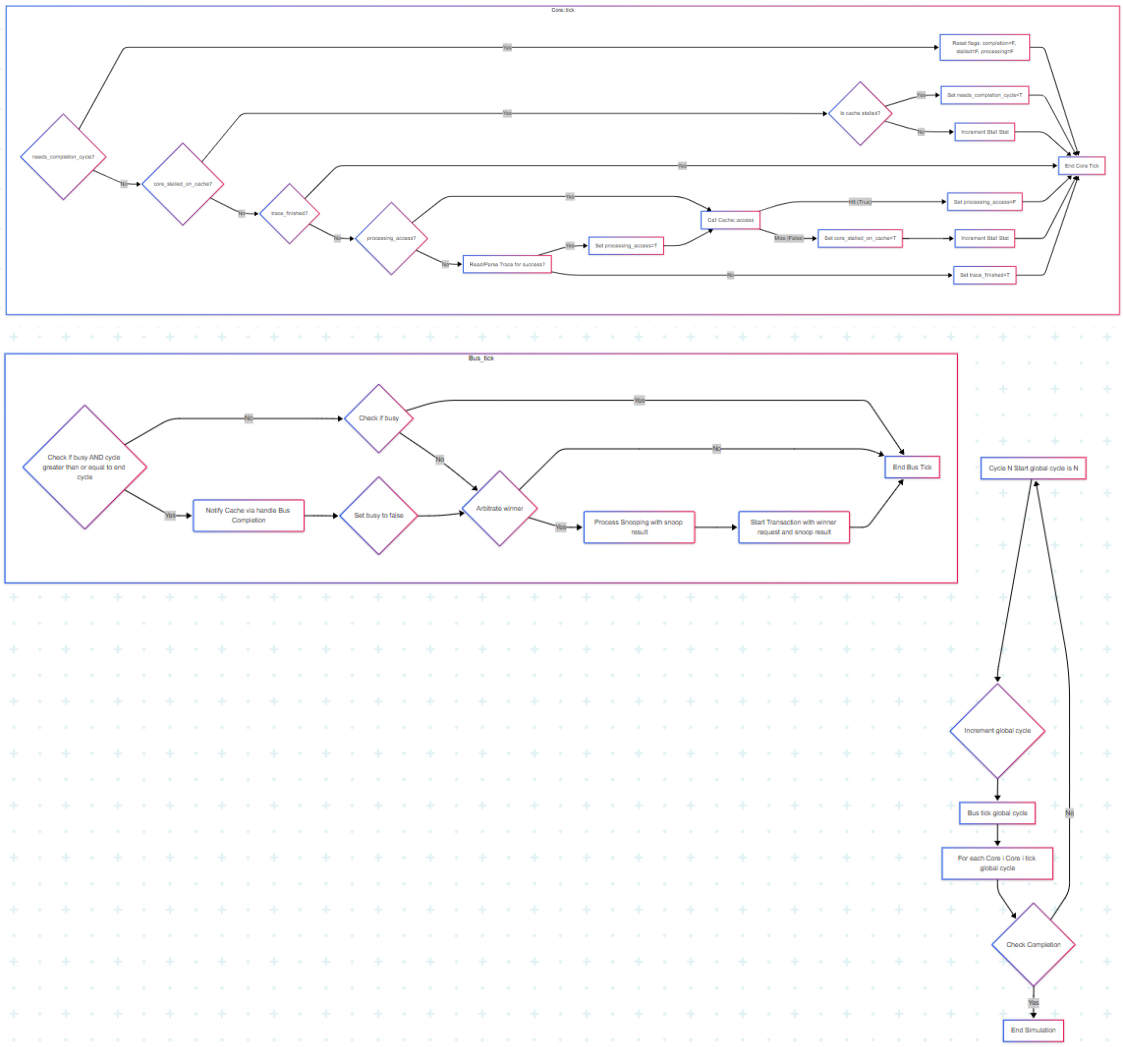

Figure 1: Simulation Cycle Flow : Each iteration of the while(true) loop represents one clock cycle.

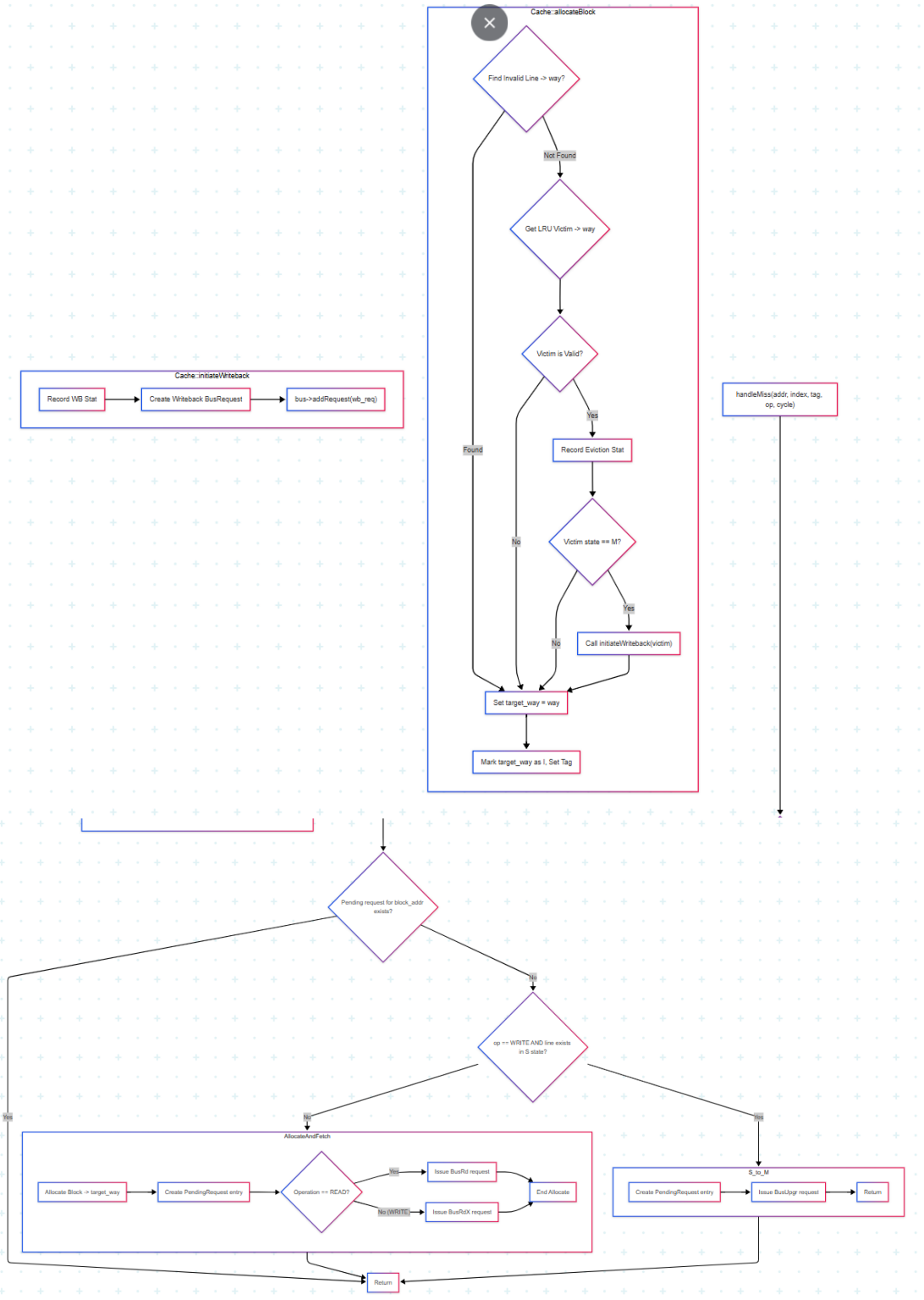

Figure 2: Miss Handling and Bus Request Generation: Called by Cache::access on a miss.

3 Experimental Analysis: Default Parameters

3.1 Experimental Setup

The default configuration used is:

- Cache size = 4 KB per core.
- Associativity = 2-way set associative.
- Block size = 32 bytes.

The simulator is executed 10 times on the same application trace files. It will not change it remains constant

3.2 Collected Statistics

The following outputs are recorded for each run:

- Instructions (read/write)
- Total cycles per core
- Cache miss rates
- Cache evictions
- Writebacks
- Bus traffic
- bus invalidations

3.3 Distribution of Outputs

Some outputs may exhibit slight variation across runs if the simulation involves any non-determinism, particularly in bus arbitration

- **Stable Outputs**:
  - Total instructions, cache accesses, cache misses, cache evictions, and writebacks remain constant across runs.
- **Potentially Variable Outputs**:
  - Stall cycles and execution cycles per core may slightly vary if the bus arbitration has any non-deterministic behavior (e.g., round-robin pointer initialized differently).

  Note: If the round-robin bus arbitration pointer is initialized differently across runs or affected by concurrent requests arriving in the same cycle, the access order may vary, introducing minor timing differences even with the same instruction streams.

4 Effect of Cache Parameters on Maximum Execution Time

4.1 Methodology

The simulator was modified to output the maximum core execution time.

The cache parameters were varied as follows:

- Cache Size: 4KB, 8KB, 16KB, 32KB (varying `-s` value).
- Associativity: 2-way, 4-way, 8-way, 16-way (varying `-E` value).
- Block Size: 32B, 64B, 128B, 256B (varying `-b` value).

Only one parameter was varied at a time, keeping others at default values.

4.2 Results and Plots

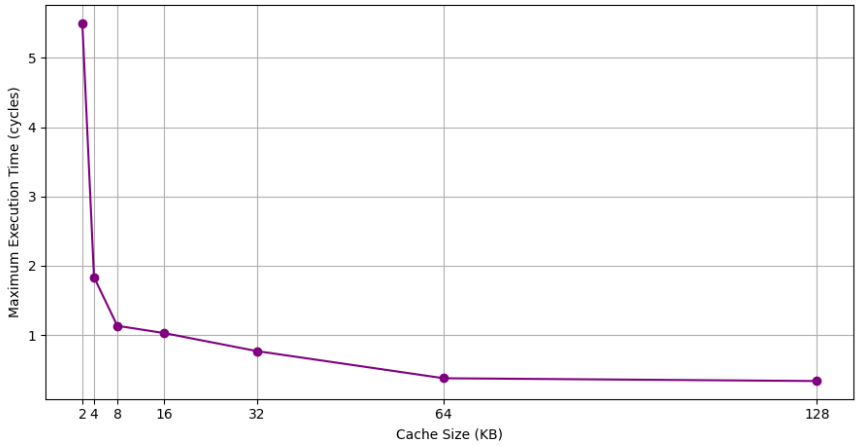

Figure 3: Maximum execution time(in terms of 10^7) vs Cache Size

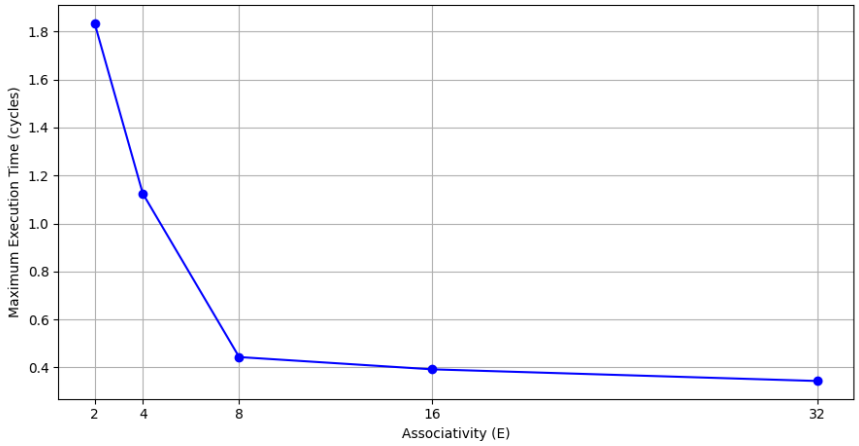

Figure 4: Maximum execution time(in terms of 10^7) vs Associativity

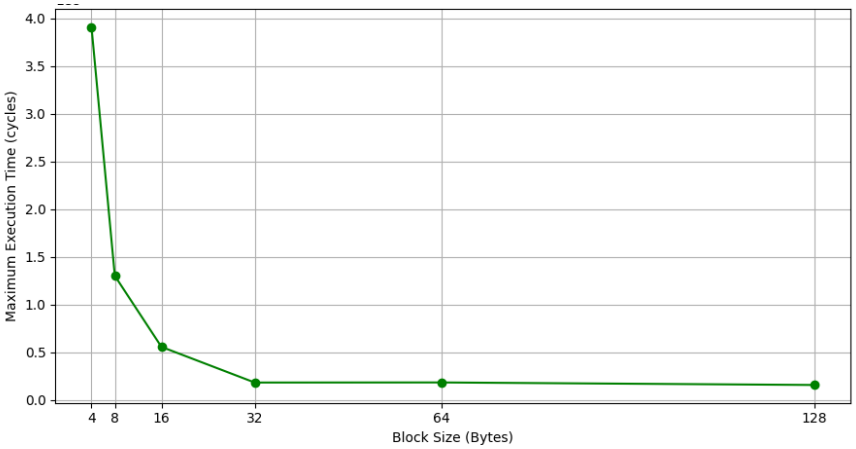

Figure 5: Maximum execution time(in terms of 10^7) vs Block Size

4.3 Observations

- Increasing cache size generally reduces maximum execution time because of fewer cache misses.
- Higher associativity helps moderately by reducing conflict misses but has diminishing returns beyond 8-way.
- Larger block sizes reduce compulsory misses but may increase false sharing, affecting performance unpredictably.