COL216: Computer Architecture
Assignment 3 Report

Neelkanth Mishra - 2023CS10022
Harshit Kansal - 2023CS10498

April 30, 2025

1 Introduction

This report outlines our approach, methodology, and implementation details for Assignment 3 of COL216. The assignment required us to simulate a L1 cache for quad-core processors, with cache coherence support.

2 Implementation

2.1 Cache Data Structure

We model each core's private cache with a single `Cache` struct that is parameterized by the global configuration bits s, b, and E. On initialization, we compute:

$$\text{sets} = 2^s, \qquad \text{blockSize} = 2^b \text{ bytes},$$

and reserve E ways per set.

Within each set we maintain four parallel arrays (all size sets $\times$ E):

- **Tag Store:** For each set and way, we keep the high-order bits of the most recently loaded memory block. We compare the incoming address tag with this array to detect hits during a lookup.
- **LRU Lists:** For each set we keep an ordered list of its E ways, from most-recently-used at the front to least-recently-used at the back. Every time a line is accessed, its entry is moved to the front; when we must evict, we remove the back entry.
- **MESI States:** Instead of maintaining separate valid and dirty bits, we store the MESI (Modified, Exclusive, Shared, Invalid) coherence state for every cache line. These states inherently capture both validity and dirtiness.

In addition, the struct carries a `stall` flag per core, which signals when that core must pause (e.g. waiting on a block fill or write-back). The `init()` routine, called once at startup, sizes these vectors to sets $\times$ E, zeroes out the tags, clears all valid/dirty bits, and seeds each LRU list with the sequence $[0, 1, \ldots, E-1]$.

2.2 Bus Transaction Structures

To coordinate communication between caches and memory in our MESI-based coherence protocol, we implement a simulated shared bus interface. This interface uses two core data structures to track coherence-related events and data movement: `BusReq` and `BusData`.

BusReq The `BusReq` structure represents a coherence request made by a core. It contains:

- **coreId:** The identifier of the core issuing the request.
- **address:** The 32-bit physical memory address involved in the transaction.
- **type:** The type of request being made, expressed as an enum with three values:
  - `BusRd`: A read request for a cache block (can be served in shared or exclusive state).
  - `BusRdX`: A read-for-ownership request (indicates intent to write, requires invalidating copies in other caches).
  - `BusUpgr`: An upgrade request to switch from shared to exclusive without refetching the data.

All requests from cores are placed in the global `busQueue`, where they are processed in FIFO order.

BusData The `BusData` structure is used to communicate the results of a bus transaction or to model memory traffic that follows a coherence event. It includes:

- **address:** The memory address associated with the cache line being transferred or written back.
- **coreId:** The ID of the core receiving the data (e.g., on a read miss).
- **write:** A boolean flag indicating whether the operation was a write (as opposed to a read).
- **writeback:** Indicates whether the line is being written back to main memory due to eviction.
- **inv:** A flag that indicates whether it was a BusUpgr signal.
- **stalls:** The number of stall cycles incurred due to this transaction.

2.3 Cache Operation Function (`run()`)

The `run()` function is the core operation handler that processes memory accesses for each core in the cache coherence implementation. It takes two parameters: a pair containing the access type (read 'R' or write 'W') and the memory address, and the core ID making the request.

Function Structure The function begins by extracting the access type and address from the input parameters. It parses the hexadecimal address and checks if the core already has a pending operation; if so, it skips processing the new request. For valid requests, it extracts the index and tag bits from the address to locate the appropriate cache line.

Read Operations For read requests ('R'), the function:

- Searches the corresponding cache set for a valid line with matching tag
- On hit: Updates the LRU order and increments the core's cycle counter
- On miss: Issues a `BusRd` request through the bus queue and stalls the core

Write Operations Write requests ('W') are handled based on the current state of the cache line:

- If line is in Modified (M) state: Proceeds locally, updates LRU, maintains M state
- If line is in Exclusive (E) state: Proceeds locally, updates LRU, transitions to M state
- If line is in Shared (S) state: Issues `BusUpgr` request, updates LRU
- On miss: Issues `BusRdX` request and stalls the core

Flow Control The function implements bus-based flow control through the core's stall flag. When a cache miss occurs or a coherence action is required, the core sets its stall flag and enqueues an appropriate bus request. The core remains stalled until the requested operation completes through the bus arbitration mechanism.

2.4 Bus Arbitration and Transaction Handling (`bus()`)

The `bus()` function simulates the shared coherence bus under the MESI protocol. Each invocation:

1. Services all pending `BusReq` requests in FIFO order (while the bus is free).
2. Processes a single `BusData` entry per cycle (modeling memory or write-back latency).

We split its logic into two phases:

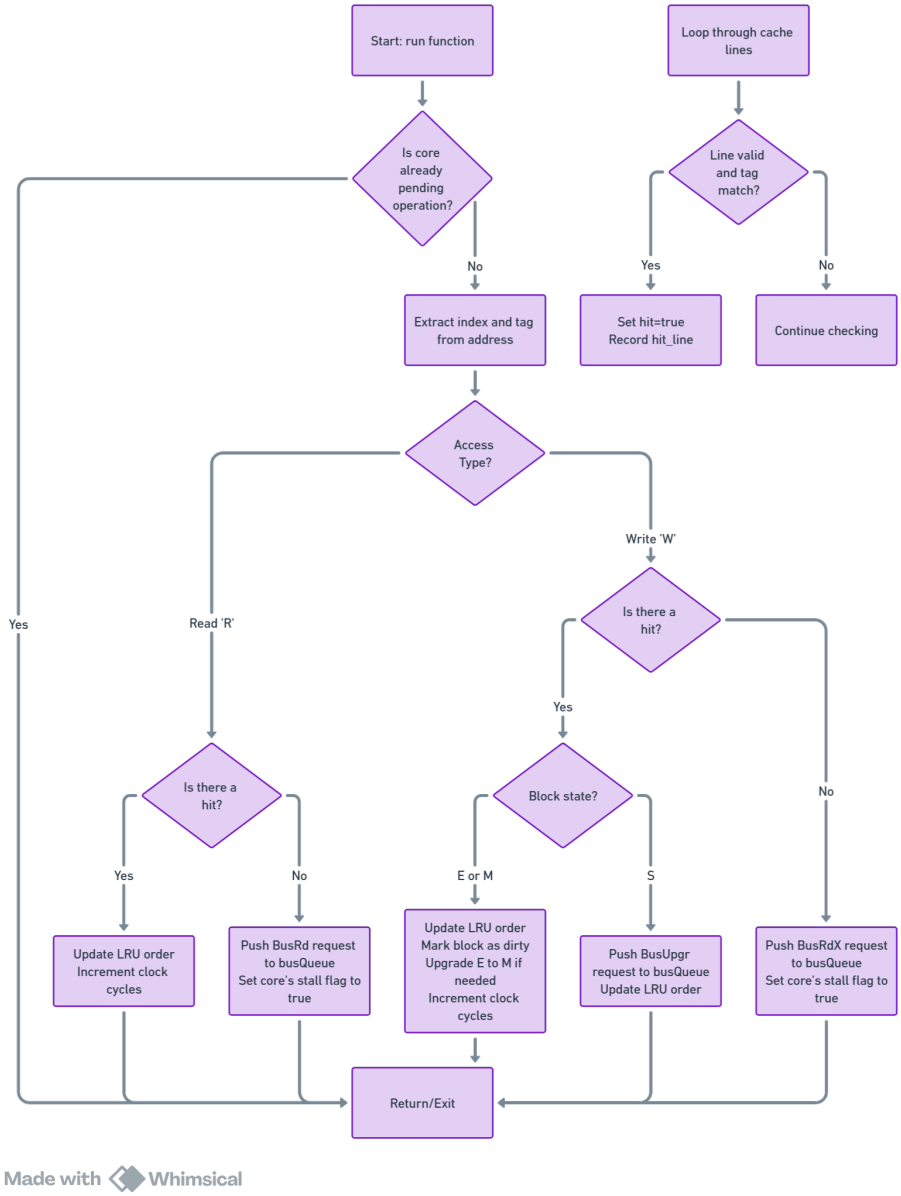

Figure 1: Flowchart depicting `run()` function

2.4.1 Bus Request Handling (BusReq)

Handles coherence operations issued by cores via the global `busQueue`. For each request:

- Decode `coreId`, `address`, and `type`.
- If the bus is busy, re-stall the requesting core and count an idle cycle.
- Otherwise, mark the bus busy, bump `total_bus_transactions` (and misses), then:

  - `BusRd`: search other caches for the block; if found in M/E, share (→S) and possibly write-back; else fetch from memory.
  - `BusRdX`: invalidate any sharers (write-back if M); then grant exclusive.
  - `BusUpgr`: invalidate sharers of a `S` line, upgrade local state to M.
- Enqueue an appropriate `BusData` entry (data transfer or write-back) and stall the requester.

2.4.2 Bus Data Handling (BusData)

Once a `BusData` entry is queued, we model its transfer or write-back:

- If `stalls>0`, decrement and wait.
- Otherwise:
  - Update `total_bus_traffic_bytes` and per-core `data_traffic_bytes`.
  - If `writeback`: finalize writeback, clear stall.
  - Else decode `isWrite`, `inv` flags:
    - **Read miss**: allocate a line via `handle_read_miss()`, set MESI to E/S.
    - **Write miss**: allocate via `handle_write_miss()`, set MESI to M.
  - Clear stall, pop `BusData`, free the bus if queue now empty.

3 Assumptions

In the design and implementation of the cache coherence simulation, we made the following assumptions to accurately model the system behavior:

- **Cache-to-Cache Transfer Stalling:** During a cache-to-cache transfer (e.g., when a block is shared or invalidated due to a coherence request), only the receiving core is stalled. The sending core, which provides the data, continues its execution without interruption unless it has its own pending operations.
- **Read Miss Latency:** For a read miss, the core first spends one cycle to detect the miss and initiate the bus request. Subsequently, it incurs a 100-cycle latency to fetch the data from main memory or another cache. The core can process a new instruction starting from the cycle immediately following the completion of this latency (i.e., cycle 102 if the miss is detected in cycle 1).
- **Bus Access Priority:** When multiple cores attempt to access the shared bus simultaneously, the core with the lower core ID is granted priority. This ensures deterministic arbitration, where lower-numbered cores (e.g., core 0) are serviced before higher-numbered cores (e.g., core 3) in case of contention.

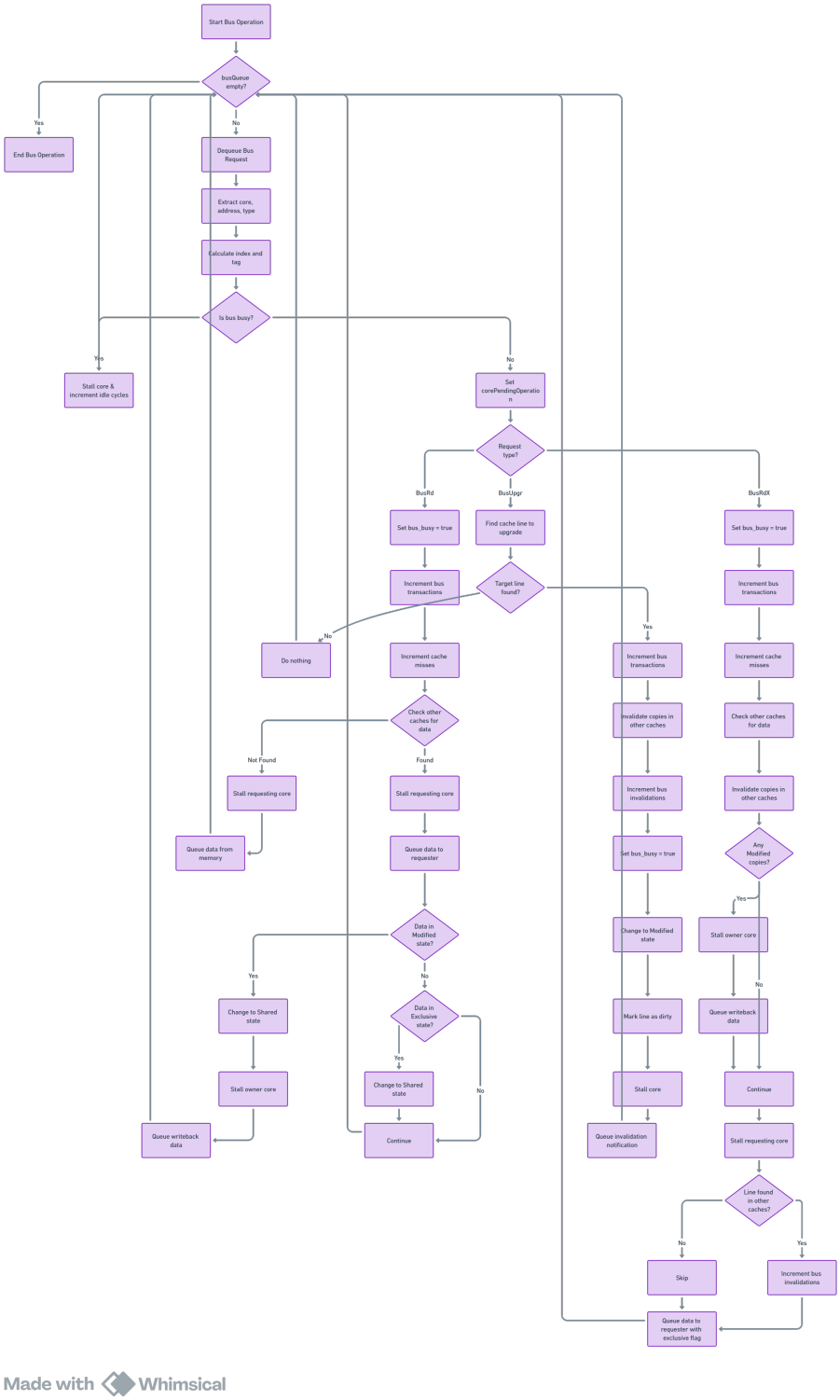

Figure 2: Flowchart depicting `BusReq`

4 Results

Since our simulator is deterministic, all outputs remain constant across multiple runs given the same inputs.

Graph 1: Impact of Varying Set-Index Bits s (with b and E Constant)

Let s be the number of set-index bits (so the number of sets $S = 2^s$), while keeping the block-offset bits b and the associativity E fixed. Under these conditions:

- **Small s (few sets).** With only $S = 2^s$ sets, many hot memory blocks contend for the same set. Since E (lines per set) is fixed, blocks evict each other frequently, causing a high conflict-miss rate. These extra misses incur long main-memory latencies, yielding the large worst-case execution time (497 k cycles at $s = 1$).
- **Moderate s.** Increasing s doubles the number of sets S, but leaves E and block size unchanged. This spreads the working set across more sets, cutting conflict misses roughly in half each time s increments by one. Hence the execution time drops sharply (e.g. from 496 k at $s = 1$ to 248 k at $s = 4$).
- **Large s (many sets).** Once S far exceeds the hot-block footprint, almost no two hot blocks map to the same set. Conflict misses become negligible, and the execution time curve plateaus (e.g. only a few thousand cycles of improvement from $s = 9$ to $s = 11$).

In short, with block size b and associativity E held constant, increasing the set-index bits s rapidly reduces conflict misses (and thus execution time) until the cache organization sufficiently accommodates the working set, after which additional sets yield diminishing returns.

Graph 2: Impact of Varying Associativity E (with b and s Constant)

With the cache's block size (2^b bytes) and number of sets (2^s) held constant, changing the associativity E (lines per set) affects only the conflict-miss behavior:

- **Very low E (direct-mapped or 2-way).** When $E = 1$ (direct-mapped), each set holds only one line, so any two blocks mapping to the same set conflict every access. This high conflict-miss rate drives up the worst-case execution time (230 k cycles at $E = 1$). Even $E = 2$ cuts miss-induced stalls roughly in half, dropping execution time to 150 k cycles.
- **Moderate E.** As you increase E from 2 to about 8, each set can accommodate more simultaneous hot blocks, sharply reducing conflict misses. Execution time falls rapidly (down to 90 k cycles by $E = 8$).
- **High E.** Beyond $E \approx 8$, most of the working-set conflicts are already resolved. Additional ways yield only marginal miss-rate improvements, so execution time plateaus (88 k cycles for $E \geq 20$).

In summary, increasing associativity E monotonically reduces conflict misses—and thus maximum execution time—but with diminishing returns once E exceeds the working-set's typical level of set contention.

Graph 3: Impact of Varying Block-Offset Bits b (with s and E Constant)

With the number of sets $S = 2^s$ and the associativity E held constant, changing the block-offset bits b (so the block size $B = 2^b$ bytes) affects two opposing factors:

- **Spatial-locality gains.** Increasing b makes each cache block larger, fetching more contiguous data on a miss. This exploits spatial locality, reducing the total number of misses and lowering execution time—hence the steep drop from $b = 1$ (720k cycles) down to the minimum around $b = 7$ (60 k cycles).
- **Conflict-and-capacity losses.** Since total cache size $C = S \times E \times B$ is effectively reduced when B grows (with S and E fixed), larger blocks mean fewer total blocks in the cache. This increases both conflict misses (hot blocks evict one another more) and capacity misses, driving execution time back up for $b > 7$ (rising toward 1M cycles at $b = 15$).

Thus the U-shaped curve arises from the trade-off between spatial-locality benefits at small-to-moderate block sizes and the miss-rate penalties of oversized blocks when b grows too large.

Graph 4: Varying Set-Index Bits s under Constant Cache Size ($C = 4096\,\mathbf{B}$) and Fixed Block Size ($b = 5$)

Here $b = 5$ fixes each block to $B = 2^5 = 32\,\mathrm{B}$, so the associativity

$$E \;=\; \frac{C}{S \cdot B} \;=\; \frac{4096}{2^s \times 32} \;=\; \frac{128}{2^s}$$

falls from $E = 64$ (at $s = 1$) down to $E = 1$ (at $s = 7$). Two opposing trends shape the curve:

- **More sets (higher s) reduce conflict misses.** As s increases, the number of sets $S = 2^s$ grows, so blocks that formerly contended for the same set are spread out. This lowers conflict misses—hence the slight drop in execution time from $s = 2$ to $s = 3$.
- **Lower associativity (higher s) increases conflict misses.** Because E halves with each extra s-bit, overly large s leads to too few ways per set. Below about $E = 16$ (i.e. $s \geq 4$), associativity loss dominates: conflict misses climb, and execution time rises sharply (especially once $E \leq 2$, at $s \geq 6$).

The net effect is a shallow "valley" around $s = 3$–4, where the balance of enough sets and enough ways best accommodates the working set. On either side—too few sets or too few ways—conflict misses increase and performance degrades.

Graph 5: Impact of Varying Set-Index Bits s and Block-Offset Bits b under Constant Cache Size ($C = 4096\,\mathbf{B}$) and $E = 2$

Because C and E are fixed, each choice of s simultaneously

$$S = 2^s, \quad B = \frac{C}{E\,S} = 2^{11-s},$$

so $s \uparrow$ more sets but smaller blocks. Two opposing effects drive the U-shaped curve:

- **Conflict-miss reduction (low $s \rightarrow$ moderate s).** Increasing s initially doubles the number of sets, spreading the working set and sharply cutting conflict misses. Execution time falls from 220 k cycles at $s = 1$ to a minimum of 115 k at $s = 4$.
- **Spatial-locality losses (moderate $s \rightarrow$ high s).** Beyond $s = 4$, block size $B = 2^{11-s}$ drops below the stride of hot accesses. Smaller blocks fetch fewer contiguous words per miss, raising compulsory misses and memory-access overhead. Execution time climbs steeply for $s \geq 5$, reaching 360 k cycles by $s = 8$ and leveling off thereafter.

Thus, under constant total cache capacity and associativity, there is an optimal balance—around $s = 4$, $b = 7$—that minimizes the combined conflict- and compulsory-miss penalties.

Graph 6: Impact of Varying Block-Offset Bits b (with Constant Cache Size $C = 4096\,\mathbf{B}$ and $s = 6$)

Fixing $s = 6$ yields $S = 2^6 = 64$ sets. As b grows, each block doubles in size, $B = 2^b$B, but the number of ways per set

$$E = \frac{4096}{64 \cdot 2^b} = \frac{64}{2^b}$$

halves. Two competing effects shape the curve:

- **Improved spatial locality (higher b).** Larger blocks bring in more contiguous data per miss, cutting compulsory misses. This effect becomes significant beyond $b = 3$ and drives the steep drop in execution time from 345 k cycles at $b = 3$ to 215 k at $b = 4$.
- **Increased conflict misses (higher b).** Because E falls from 32 at $b = 1$ down to 1 at $b = 6$, reducing the number of lines per set raises conflict misses. You can see a small bump in execution time at $b = 2$ (when E drops from 32 to 16) and a slower decline for $b \geq 5$ as spatial gains begin to saturate while associativity reaches its minimum.

Overall, with s fixed, larger block sizes eventually dominate—so once you cross the spatial-gain threshold (around $b = 4$), execution time falls continuously, despite the decline in associativity.

Graph 7: Impact of Varying Associativity E (with Constant Cache Size $C = 4096\,\mathrm{B}$ and $b = 5$)

With the block size fixed at $B = 2^5 = 32\,\mathrm{B}$ ($b = 5$) and the cache size constant at $C = 4096\,\mathrm{B}$, the number of sets S varies inversely with associativity E:

$$S = \frac{C}{E \cdot B} = \frac{4096}{E \cdot 32} = \frac{128}{E}, \quad s = \log_2 S = \log_2 \left(\frac{128}{E} \right).$$

As E increases from 1 (at $s = 7$) to 64 (at $s = 1$), the number of sets $S = 2^s$ decreases exponentially. This trade-off between associativity and the number of sets creates two competing effects on conflict misses:

- **Increased associativity reduces conflict misses within sets.** At low E, such as $E = 1$ (direct-mapped, $s = 7$), each set can hold only one cache line, leading to frequent conflict misses when multiple blocks map to the same set. Increasing E allows more lines per set, reducing conflicts within each set. For example, moving from $E = 1$ to $E = 4$ (corresponding to $s = 5$) lowers execution time by accommodating more hot blocks in each set, reducing evictions.
- **Decreased number of sets increases conflict misses across sets.** As E increases, the number of sets $S = 128/E$ decreases, causing more memory blocks to map to fewer sets. At high E, such as $E = 64$ (where $s = 1$, $S = 2$), the small number of sets leads to increased contention, as many blocks compete for the same set, raising conflict misses and execution time.

The performance curve shows a shallow minimum around $E = 8$ to $E = 16$ (corresponding to $s = 4$ to $s = 3$), where the benefits of increased associativity and a sufficient number of sets are balanced. At lower E (e.g., $E = 1$), insufficient associativity causes high conflict misses within sets. At higher E (e.g., $E = 64$), the reduced number of sets dominates, increasing conflict misses across sets. This trade-off results in optimal performance when E and S are balanced for the working set.

Interesting Traces

Trace 1: False Sharing We constructed a trace to analyze cache coherence behavior when a block is repeatedly read by one core (Core 0), while another core (Core 1) continuously writes to a nearby memory address. The trace is designed to create frequent coherence actions and demonstrate contention over the bus.

Core 0 Trace:

```
R 0x0
R 0x0
R 0x2
R 0x0
R 0x0
R 0x4
```

```
R 0x0
R 0x0
R 0x6
```

Core 1 Trace: Multiple writes to address `0x1`, continuously:

```
W 0x1
W 0x1
...
(repeated arbitrary number of times)
```

The block size is 2 bytes, so `0x0` and `0x1` map to the same block. Core 1 repeatedly writes to `0x1`, bringing the block into the `M` (Modified) state, which forces Core 0's subsequent reads to miss and trigger coherence actions.

Each pair of `R 0x0` in Core 0 is structured such that: - The first read is a **miss**, since Core 1 holds the block in `M` state. - The second read is a **hit**, allowing Core 1 to perform a `BusUpgr`, otherwise Core 0 would gain bus access. - Additional reads to increasing addresses keep Core 0 active and ensure staggered stalls, simulating competition for memory access.

Observation:

- Core 0 has nine reads and six misses, which aligns with the pattern—every first `R 0x0` in each pair results in a miss.
- Core 1 causes six bus invalidations, which matches the number of unique first accesses to `0x0` by Core 0.
- Each pair of `R 0x0` in Core 0 forces Core 1 to perform a `BusUpgr` (bus upgrade) to invalidate Core 0's copy, ensuring exclusive access for write-back.
- Core 0's repeated reloading of the same block demonstrates how read misses can be forced repeatedly due to external invalidations, despite spatial locality.

Trace 2: Sequential Reads with Serialized Access In this trace, both Core 0 and Core 1 execute identical sequences of read instructions to the same memory addresses. The block size is 2 bytes.

Core 0 and Core 1 Trace:

```
R 0x0
R 0x2
R 0x4
R 0x6
R 0x8
R 0x10
R 0x12
R 0x14
R 0x16
```

Observation:

- Core 0 always accesses the bus first and must fetch data from memory, leading to a high total execution time (909 cycles) but no idle time.
- Core 1 is frequently stalled, waiting for the bus. When it gains access, it receives data from Core 0, resulting in very low execution time (18 cycles) but high idle time (909 cycles).
- This trace highlights how fixed bus arbitration (favoring Core 0) significantly penalizes Core 1's responsiveness while overloading Core 0 with memory fetch responsibilities.

Trace 3: Impact of Array Padding on Cache Hit Rate The following C code computes the dot product of two vectors:

```
float sum = 0.0;
int i;

for (i = 0; i < 8; i++)
    sum += x[i] * y[i];
return sum;
```

Assumptions:

- Each `float` is 4 bytes.
- Cache block size is 16 bytes (holds 4 floats).
- 2-set cache, total cache size = 32 bytes.

Memory Access Trace (Original `float x[8], y[8]`):

```
R 0x00   (x[0])
R 0x20   (y[0])
R 0x04   (x[1])
R 0x24   (y[1])
R 0x08   (x[2])
R 0x28   (y[2])
R 0x0C   (x[3])
R 0x2C   (y[3])
R 0x10   (x[4])
R 0x30   (y[4])
R 0x14   (x[5])
R 0x34   (y[5])
R 0x18   (x[6])
R 0x38   (y[6])
R 0x1C   (x[7])
R 0x3C   (y[7])
```

This leads to many conflict misses due to interleaved access patterns to `x` and `y` that map to the same cache set.

Modified with Padding: `float x[12];` adds spacing to `x`, changing its memory layout. The resulting access pattern:

Memory Access Trace (Padded `x[12]`):

```
R 0x00   (x[0])
R 0x30   (y[0])
R 0x04   (x[1])
R 0x34   (y[1])
R 0x08   (x[2])
R 0x38   (y[2])
R 0x0C   (x[3])
R 0x3C   (y[3])
R 0x10   (x[4])
R 0x40   (y[4])
R 0x14   (x[5])
R 0x44   (y[5])
R 0x18   (x[6])
R 0x48   (y[6])
R 0x1C   (x[7])
R 0x4C   (y[7])
```

Observation:

- With padding, accesses to `x` and `y` no longer map to the same cache set, reducing conflict misses.
- This modification increases the cache hit rate from nearly 0% to 75%.

Simple Trace to Check Correctness

Trace Description This simple trace is designed to validate the correctness of the coherence protocol and cache state transitions by simulating a minimal interaction between two cores on the same memory block.

Core 0 Trace:

```
R 0x00000000
R 0x00000000
W 0x00000000
```

Core 1 Trace:

```
R 0x00000000
R 0x00000000
R 0x00000000
R 0x00000800
```

Trace Explanation:

- Core 0 begins with a read to `0x0`, which results in a read miss and the block is loaded into its cache in the `E` or `S` state.
- The second read by Core 0 is a hit.
- Then, Core 1 reads `0x0` causing a read miss. It takes the block from Core 0, leading to both caches holding it in the `S` state.
- Core 1's second read is now a hit.
- Core 0 then writes to `0x0`, which is a write hit and causes an invalidation of Core 1's shared copy. The block in Core 0 transitions to the `M` state.
- Core 1 attempts another read to `0x0`, which is a read miss. It forces a writeback from Core 0 and fetches the block, again transitioning to `S`.
- The final read by Core 1 is to address `0x800`, mapping to a new block. This causes a read miss and leads to an eviction due to index conflict.
- As a result:
  - Core 0 sees 1 miss, 1 writeback, and 1 invalidation.
  - Core 1 sees 3 misses, 1 eviction.
  - Data traffic arises from memory loads, cache-to-cache transfers, and memory writebacks.

Simulation Output: Core 0 Statistics:

- Total Instructions: 3
- Total Reads: 2
- Total Writes: 1
- Total Execution Cycles: 103
- Idle Cycles: 16
- Cache Misses: 1
- Cache Miss Rate: 33.33%
- Cache Evictions: 0
- Writebacks: 1
- Bus Invalidations: 1
- Data Traffic (Bytes): 160

Core 1 Statistics:

- Total Instructions: 4
- Total Reads: 4
- Total Writes: 0
- Total Execution Cycles: 136
- Idle Cycles: 202
- Cache Misses: 3
- Cache Miss Rate: 75.00%
- Cache Evictions: 1
- Writebacks: 0
- Bus Invalidations: 0
- Data Traffic (Bytes): 96

Core 2 Statistics:

- Total Instructions: 0
- Total Reads: 0
- Total Writes: 0
- Total Execution Cycles: 0
- Idle Cycles: 0
- Cache Misses: 0
- Cache Miss Rate: 0.00%
- Cache Evictions: 0
- Writebacks: 0
- Bus Invalidations: 0
- Data Traffic (Bytes): 0

Core 3 Statistics:

- Total Instructions: 0
- Total Reads: 0

- Total Writes: 0
- Total Execution Cycles: 0
- Idle Cycles: 0
- Cache Misses: 0
- Cache Miss Rate: 0.00%
- Cache Evictions: 0
- Writebacks: 0
- Bus Invalidations: 0
- Data Traffic (Bytes): 0

Overall Bus Summary:

- Total Bus Transactions: 5
- Total Bus Traffic (Bytes): 192

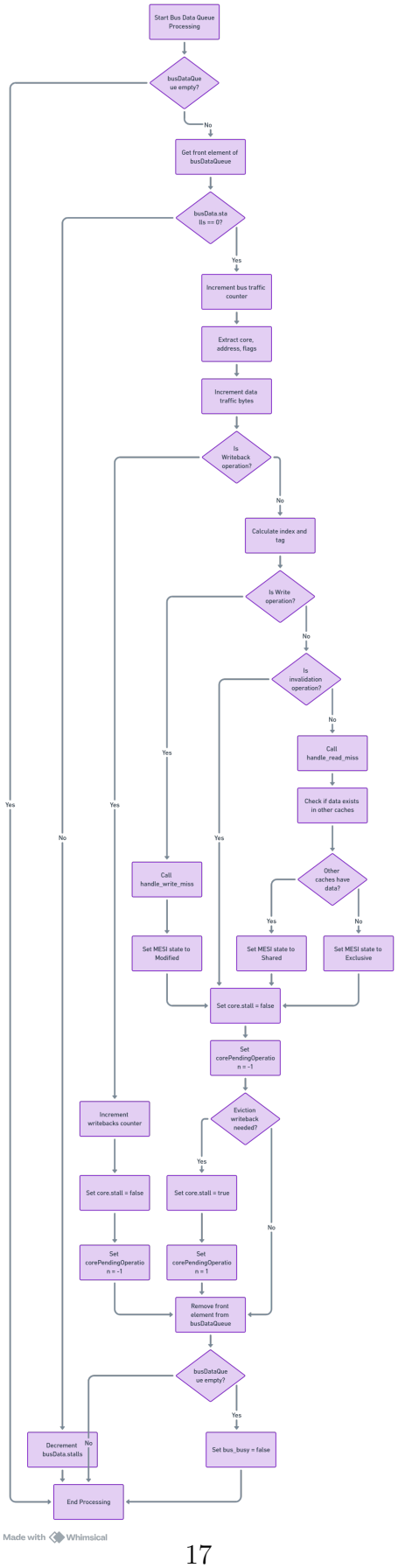

Figure 3: Flowchart depicting `BusData`

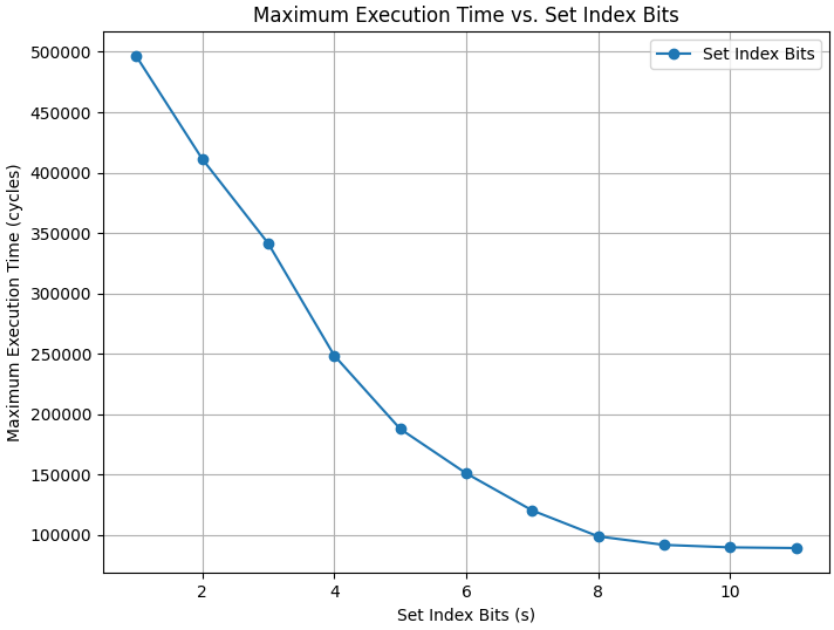

Figure 4: Maximum execution time vs. set index bits

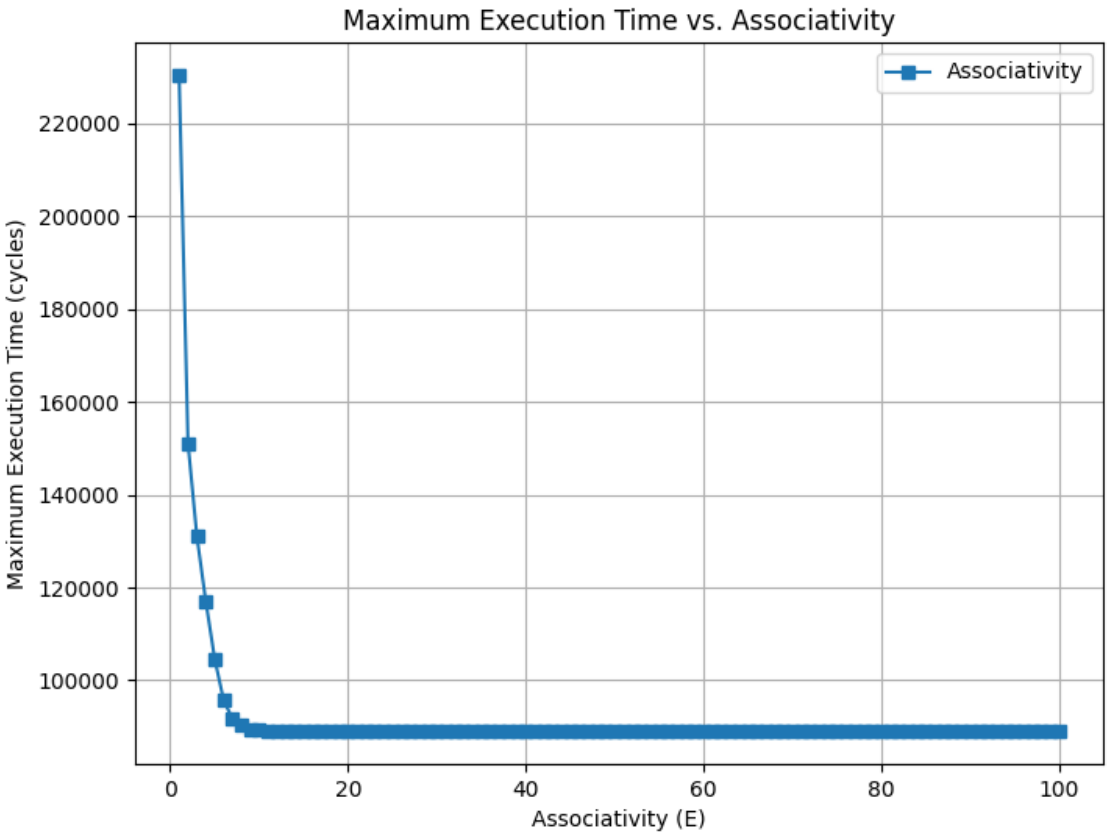

Figure 5: Maximum execution time vs. associativity E, for fixed block-offset bits b and set-index bits s.

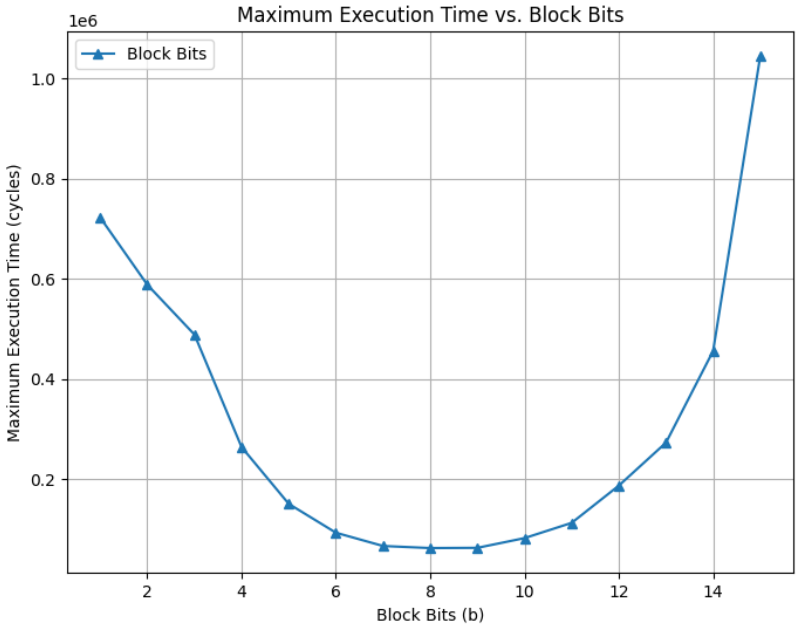

Figure 6: Maximum execution time vs. block index bits

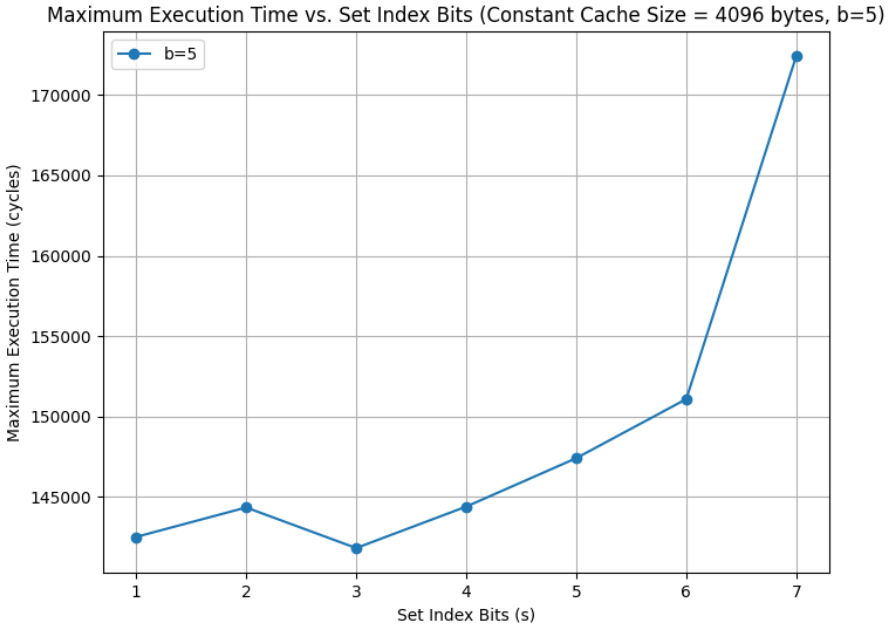

Figure 7: Constant cache size (varying set index bits and associativity bits)

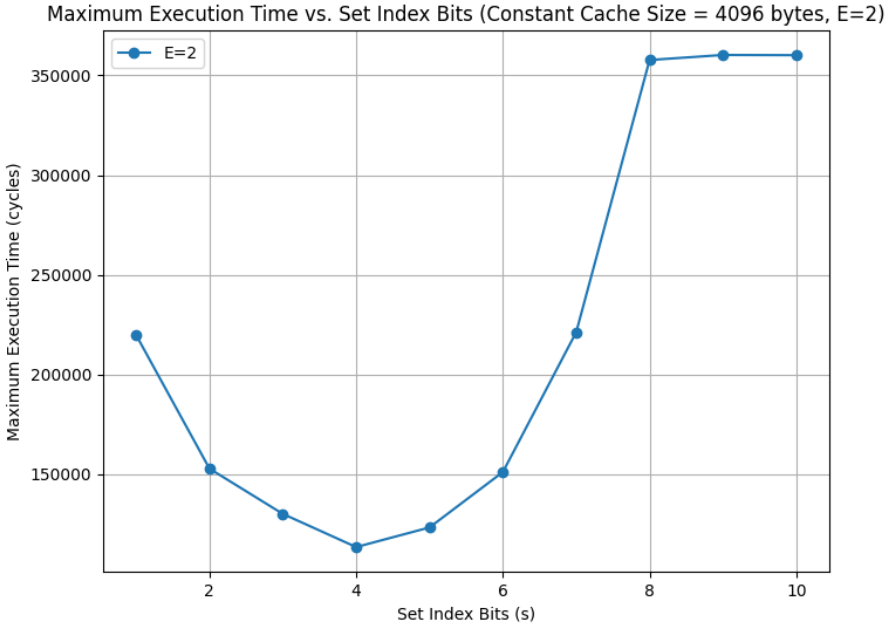

Figure 8: Constant cache size (varying set index bits and block bits)

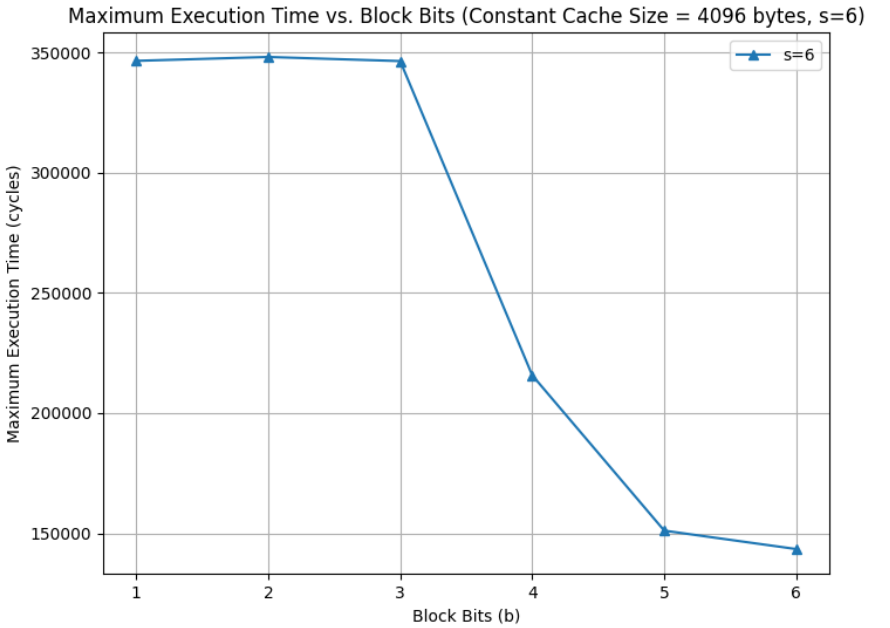

Figure 9: Constant cache size (varying block bits and associativity bits)

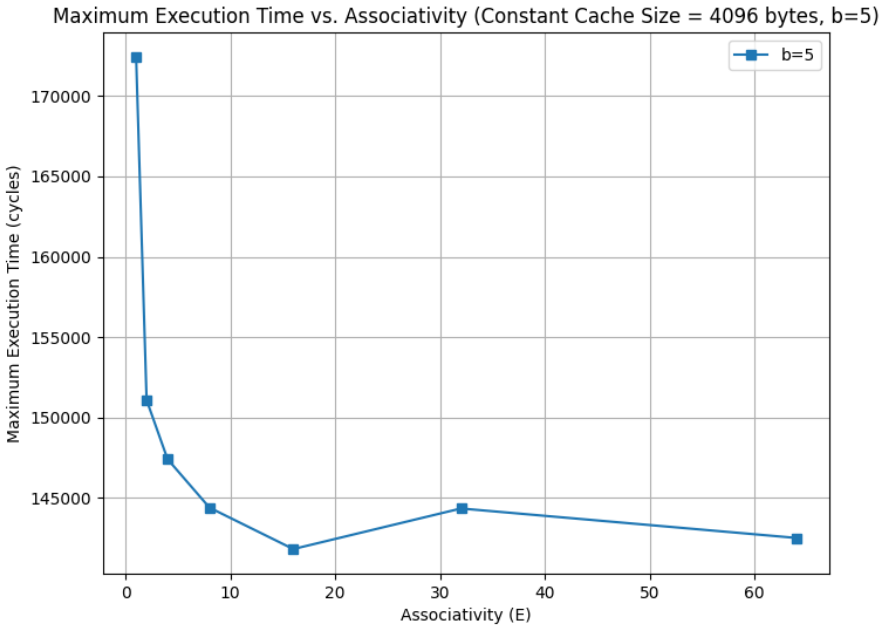

Figure 10: Constant cache size (varying set index bits and associativity bits)