Project Report Summary

Assignment 3

Submitted by:
Amber Agarwal (2023CS50625)
Anoop Singh (2023CS10459)

Supervisor:
Prof. Rijurekha Sen

Institution Name
IIT Delhi
Department Name
Computer Science and Engineering

Contents

1 Sections **2**

1.1 Main Classes and Structures 2
1.2 Data Structures 2
1.3 Rules Followed for the Assignment 2
1.4 Graphs and Analysis 4
1.5 Observations 6
1.6 Flowchart 6
1.7 False Sharing 8
1.8 Test Traces Analysis 9

1 Sections

1.1 Main Classes and Structures

- **Cache**: This is the core class representing a single cache instance. Each processor has its own `Cache` object that stores tag and data arrays, tracks statistics, and implements logic for handling hits, misses, invalidations, and bus transactions. Important methods include:
  - `hit_or_miss`, `read_hit`, `write_hit`
  - `read_miss`, `write_miss`
  - `handle_bus_transaction_completion`
- **Bus**: A central shared communication medium simulating a snooping bus. It includes fields like `busy`, `cycle_remaining`, `transaction_type`, etc., to manage bus contention, invalidations, and cache-to-cache transfers.
- **Statistics**: A structure holding performance counters such as number of reads, writes, cache misses, idle cycles, data traffic, etc.
- **Bits**: Used to store parsed tag, index, and offset bits from an address.
- **Global Simulation**: The `main` function sets up four `Cache` instances (for a 4-core simulation), a shared `Bus`, and processes traces in a cycle-by-cycle fashion until all caches finish processing their instructions.

1.2 Data Structures

- `vector<vector<CacheLineMeta>> tag_array`: A 2D vector storing tag, state (MESI), and timestamp for each cache line.
- `vector<vector<vector<int>>> data_array`: Placeholder for data blocks per set and way.
- `vector<pair<operation, string>> trace_data`: Holds parsed memory accesses (read/write and address) from trace files.

1.3 Rules Followed for the Assignment

- The MESI cache coherence protocol is used to manage consistency across caches.
- In case multiple cores request bus access simultaneously, the core with the lower ID is given priority.
- A single shared bus is assumed for the entire execution. It handles:
  - Data transfer between cache and memory.
  - Data transfer between caches (cache-to-cache).

  - Broadcasting of coherence signals.

- If the bus is busy, any core that wishes to place a request must wait until the bus becomes free.

- We assume:
  - The sender does not stall and immediately updates its state after placing data on the bus.
  - The receiver stalls until it receives the required data and then updates its state.

- Execution cycles include:
  - Time when a core actively processes its instruction.
  - Time spent waiting for data via the bus.

- Idle cycles occur when:
  - A core is ready to execute an instruction but must wait for the bus.

- No increment:
  - A core has finished all its instructions (no execution or idle cycles are counted thereafter).

- Cache evictions follow the LRU (Least Recently Used) policy:
  - If an invalid block exists, it is replaced directly.
  - Otherwise, the least recently used block is evicted.
  - If the evicted block is in the `M` (Modified) state, it is first written back to memory.
  - The eviction counter is incremented for each such replacement.

- `Writeback` counter is incremented whenever a block in the `M` state is sent to memory:
  - During cache eviction.
  - When another core's read miss fetches data from this cache.
  - When a core encounters a write miss and another core has the modified block.

- `BusInvalidation` counter is incremented when:
  - A core experiences a write miss or write hit.
  - Another cache has the block in a state other than `I` (Invalid).

- `BusTraffic` includes:

- All data transmitted to or from the bus.

- **Bus statistics** include:
  - `Total bus transactions` — count of data or signal transmissions over the bus.
  - `Total data traffic` — sum of all bytes transferred through the bus.

1.4 Graphs and Analysis

We have generated the graphs by varying four different aspects of the cache configuration to observe their impact on performance:

1. **Varying the Number of Sets in the Cache:** We increase the number of sets. This typically results in more cache hits due to improved address mapping, thereby reducing the number of execution cycles. This trend is clearly visible in the outputs for all four cores, as shown in Figure 1.

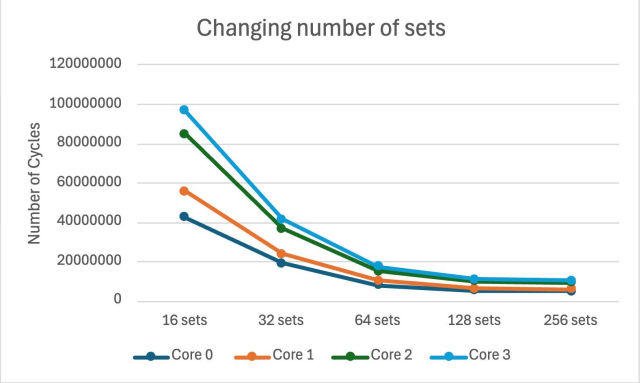

Figure 1: Performance variation with different numbers of cache sets

2. **Varying the Associativity of the Cache:** Increasing the associativity generally allows for better conflict resolution in set-indexed caches, which can improve cache hit rates. However, higher associativity also increases complexity and access latency. The corresponding performance implications can be seen in Figure 2.

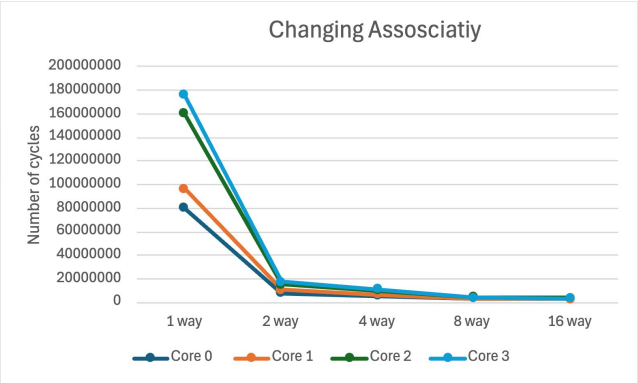

Figure 2: Performance variation with different cache associativity

3. **Varying the Offset Bits in a Block:** Modifying the number of offset bits changes the block size. Larger blocks can reduce miss rates for sequential access patterns. The resulting effects on performance are illustrated in Figure 3.

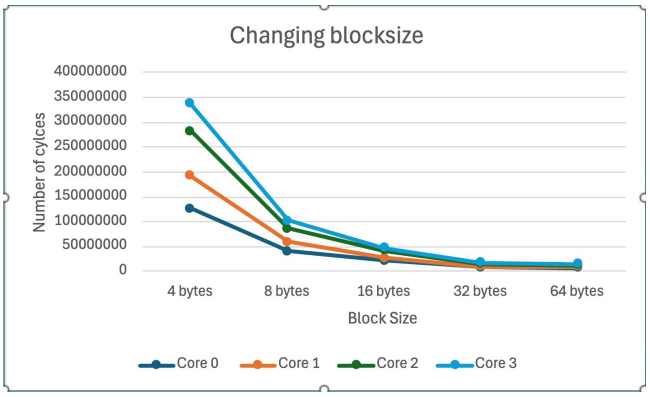

Figure 3: Performance variation with different block offset sizes

4. **Toggling Between Number of Sets and Associativity (with Fixed Cache Size):** In this case, we maintain the overall cache size constant and vary the configuration by adjusting the number of sets and the associativity accordingly. This trade-off allows us to analyze which combination provides optimal performance under a constrained memory budget. We see that the cycles first decrease and then increase. This behavior is due to the fact that although we reduce conflict misses, on reducing the number of sets, the blocks pointing to different sets encounter misses. The comparative results are presented in Figure 6.

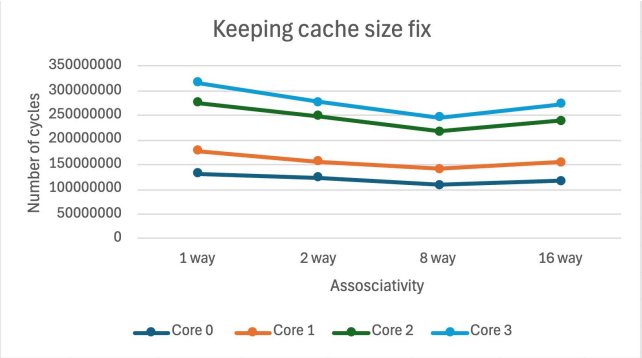

Figure 4: Performance variation with fixed cache size under different set/associativity configurations

1.5 Observations

- Across all the performance graphs, we observe that **Core 3 consistently incurs the highest number of execution cycles**, resulting in the longest completion time among all cores.
- This outcome is primarily due to the bus arbitration policy adopted in our simulator, which **prioritizes cores based on ascending core ID** in the event of simultaneous bus access requests.
- As a result, Core 0 always receives the highest priority, followed by Cores 1, 2, and finally 3. This fixed ordering causes Core 3 to frequently wait the longest for bus access, leading to increased idle time and total execution cycles.
- The assignment also required us to run the simulator 10 times using the same application and default parameters, and report the distribution of outputs across these runs.
- However, since our simulator employs a **deterministic conflict resolution strategy**, the output remains identical across all runs. Consequently, no variation or statistical distribution is observed in the results.

1.6 Flowchart

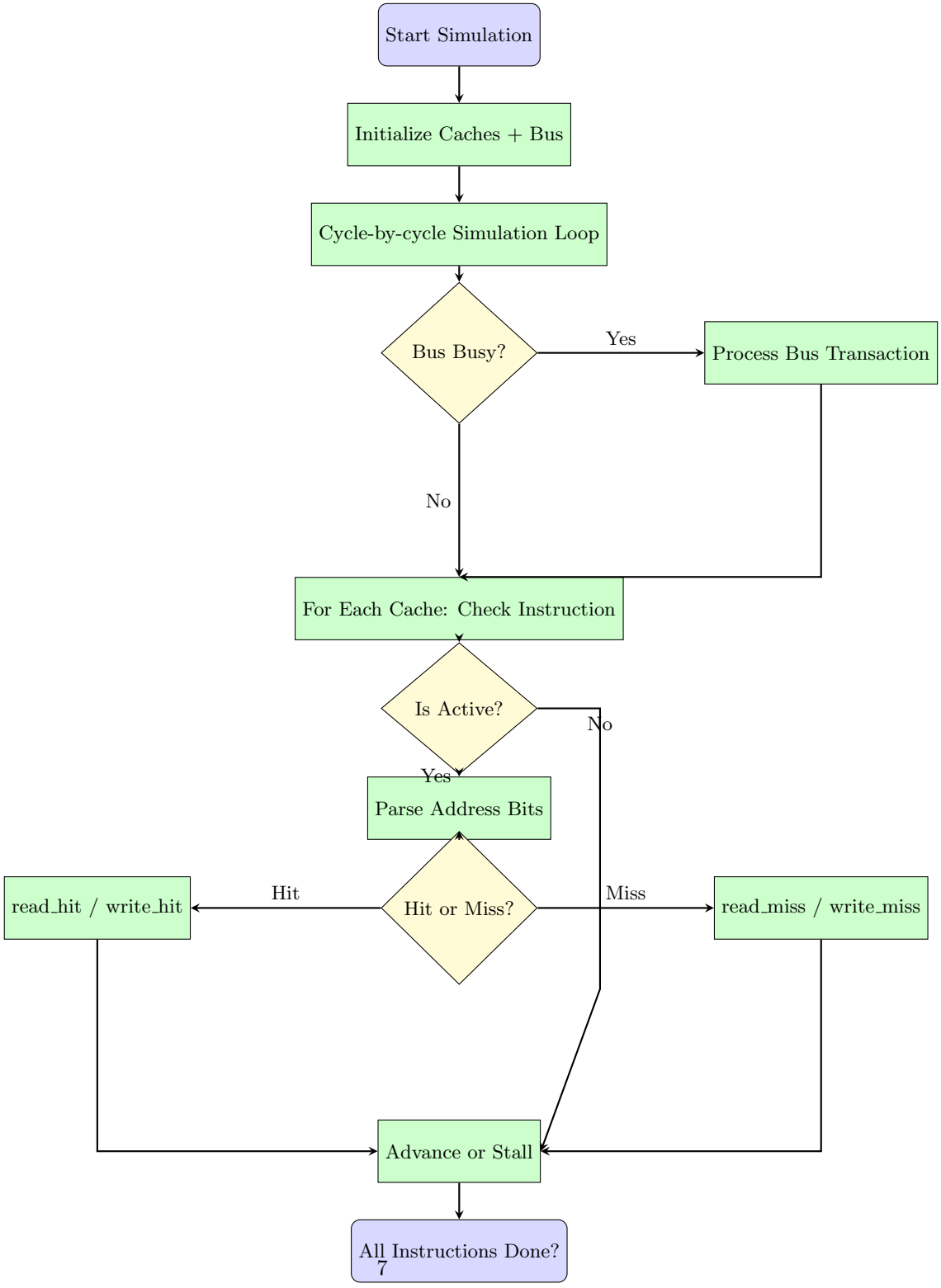
Start Simulation
Initialize Caches + Bus
Cycle-by-cycle Simulation Loop
Bus Busy?
Yes
Process Bus Transaction
No
For Each Cache: Check Instruction
Is Active?
No
Yes
Parse Address Bits
Hit or Miss?
Hit
read_hit / write_hit
Miss
read_miss / write_miss
Advance or Stall
All Instructions Done?

Figure: Simulation control flow between bus and cache processors

1.7 False Sharing

We have shown the false sharing by comparing two different apps doing identical operations but one causes false sharing while other does not.

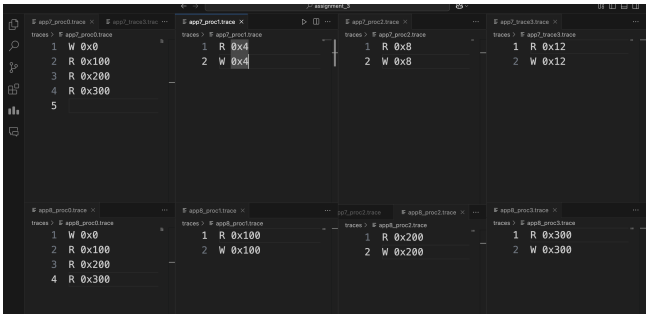

Figure 5: False Sharing Traces

In both the apps in trace0 we have did 4 reads which all will get miss consuming the accounting for 400 cycles in both the apps.

Then from trace1 onwards each trace does a read and then a write on some address, in app1 all the addressees are from the same cache line so each one will be a cache miss and data will come from the cache0 via bus , in app2 all the adress belong to different cachelines in core 0 so here also the data come from cache 0 via bus.

Then there is a local write hit in each process so one would expect similar number of cycles in both the apps, but the here comes the false sharing in app1 all the adressess are from the same cache line so when one core has a read miss then it has to take data from the previous core but the previous core has the same cache line in modified state which has to be written back into memory causing 100 cycles delay, this happen for all the three cores leading to a differnce of 300 cycles as can bee seen in the output.

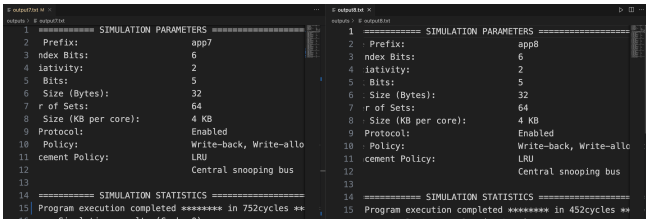

Figure 6: False Sharing Output

1.8 Test Traces Analysis

- **App3:** In this trace, every core writes to the same memory location. On a write miss, a core may find the block in another cache in the `M` (Modified) state. The owning cache must first write the data back to memory before the requesting cache can fetch it. This results in additional delays, leading to approximately **300 extra cycles** compared to **App5**, as shown in `output9.txt`.

- **App4:** This trace runs on a cache configuration with only one set. Initially, Core 0 reads from memory, followed by Core 1 writing to a different address. Then, Core 1 writes again to `0x0`, which was previously loaded by Core 0. This causes a write miss and triggers a bus transaction. Core 1 invalidates Core 0's block and prepares to write. However, due to the single-set constraint and presence of a block in `M` state, Core 1 must first write back the existing block before writing its own data. This showcases both invalidation and forced write-back due to limited associativity.

- **App5:** In this trace, each core reads from a different block. As a result, each core requires bus access during its execution phase. Since the bus is shared, this creates a **sequential execution** scenario. The total number of cycles required is **401**, as shown in `output5.txt`.

- **App6:** This test focuses on a single core performing reads from different addresses. Each read results in a miss and fetches data from memory. Other cores remain idle throughout, with **zero idle and execution cycles**. This demonstrates isolated execution. Refer to `output6.txt` for results.

- **App7 and App8:** These are analyzed under the **False Sharing** subsection of the report. App7 suffers from false sharing due to writes to addresses within the same cache line, causing unnecessary invalidations and write-backs, while App8 avoids this by writing to different cache lines.

- **App9:** This trace features reads to the **same memory location** from all cores. Since the data is already present in one core's cache, **cache-to-cache transfers** occur, reducing memory access time. As a result, the execution time drops significantly to **149 cycles** (compared to 401 in App5). See `output9.txt`.

- **App10:** All cores perform writes to the same memory location. On a write miss, if another cache holds the block in `M` state, it must first write it back. Then the requesting cache fetches the data from memory. This results in **300 additional cycles** compared to App5, as shown in `output10.txt`.