Cache Assignment Report

1 Design Decisions

This section provides an in-depth overview of the implementation of our multicore cache simulator supporting the MESI coherence protocol. The system models per-core private L1 caches, a shared system bus, and core-generated memory access traces to simulate cache behavior in a multicore environment.

1. Class L1Cache

This class models a private L1 cache per core. It implements MESI-based coherence, cache lookup logic, and interaction with the shared bus.

Key Structures and Members:

- `vector<CacheSet> sets`: All cache sets. The number of sets is 2^s.
- `int s, E, b, S, B, W`: Cache configuration parameters:
  - `s`: Number of set index bits ($S = 2^s$).
  - `E`: Associativity (lines per set).
  - `b`: Number of block offset bits ($B = 2^b$).
  - `W`: Number of 4-byte words per block ($B/4$).
- `shared_ptr<Bus> bus`: Shared pointer to the central bus.
- `uint64_t accesses, hits, misses, evictions, writes, reads, etc.`: Performance counters.
- `bool has_pending_request, is_waiting_for_bus`: Track if the cache is currently stalled.
- `uint32_t pending_address`: Address currently waiting on a bus transaction.
- `int pending_response`: Response code from bus (used to determine MESI state after snooping).

Important Methods:

- `accessMemory()`: Handles read/write access from the core, initiates bus requests on misses.
- `continuePendingAccess()`: Finalizes pending transactions once the bus is free.
- `snoopBus()`: Listens to bus transactions and updates cache lines accordingly.
- `printStats()`: Outputs detailed runtime statistics and cache state.

2. Class Bus

Models the communication medium shared among all cores. Handles queuing and arbitration of memory transactions.

Key Structures and Members:

- `vector<L1Cache*> caches`: List of registered L1 caches.
- `queue<BusTransaction> pending_requests`: FIFO queue of memory transactions (read/write/invalidate).
- `uint64_t bus_busy_until`: Clock cycle until which the bus is considered busy.
- `int last_serviced_core`: Core ID of the last serviced bus request.

Important Methods:

- `addPendingRequest()`: Adds a transaction to the queue.
- `processNextRequest()`: Handles the next request if the bus is free.
- `invalidateAll()`: Sends invalidation signals to all caches except the source.
- `isBusy()`, `getBusAvailableCycle()`: Query current bus status.

3. Class CacheSimulator

Manages the simulation lifecycle.

Key Members:

- `vector<unique_ptr<L1Cache>> caches`: The four L1 cache instances.
- `shared_ptr<Bus> bus`: Shared bus object.
- `vector<ifstream> traceFiles`: Input trace streams per core.
- `vector<bool> coreFinished`: Flags indicating if each core has finished execution.
- `vector<string> currentOps`: Current memory operation per core.
- `vector<uint32_t> currentAddrs`: Current memory address per core.

Important Methods:

- `runSimulation()`: Core simulation loop.
- `readNextInstruction()`: Fetches the next memory operation for a core.
- `printFinalStats()`: Aggregates and outputs final statistics.

4. Struct CacheSet

Represents a single set in the cache, which contains multiple `CacheLine` objects (equal to associativity).

Key Members:

- `vector<CacheLine> lines`: Cache lines within the set.
- `uint32_t lru_timestamp`: Timestamp used to implement LRU replacement.

Important Methods:

- `updateLRU(int lineIndex)`: Updates the LRU counter when a line is accessed.
- `findLRULine()`: Identifies the least recently used line for eviction.

5. Struct CacheLine

Models an individual cache line (or block) within a set.

Key Fields:

- `bool valid, dirty`: Line status flags.
- `uint32_t tag`: Tag bits from address used for lookup.
- `CacheLineState state`: MESI state of the line.
- `uint32_t lru_counter`: Used for LRU eviction.
- `vector<uint8_t> data`: Simulated block data (not used for correctness, mainly for realism).

6. Struct BusTransaction

Encapsulates a memory operation request on the bus.

Fields:

- `BusOp operation`: Enum indicating operation type (e.g., `BUS_RD`, `BUS_RDX`).
- `uint32_t address`: Memory address.
- `int sourceCore`: ID of the core that issued the transaction.
- `bool serviced`: Indicates whether any cache responded to the transaction.

7. Address Breakdown and Cache Access

- Addresses are decomposed into `tag`, `set index`, and `block offset`.
- Functions like `getTag()`, `getSetIndex()`, and `getBlockOffset()` handle this.
- Each memory access leads to a cache lookup, and if necessary, a bus transaction and state update.

8. LRU Replacement Strategy

Within each `CacheSet`, the LRU strategy is implemented via a global counter that updates on every access. The line with the lowest counter value is considered least recently used and is evicted when needed.

2 Flowchart of Important Functions

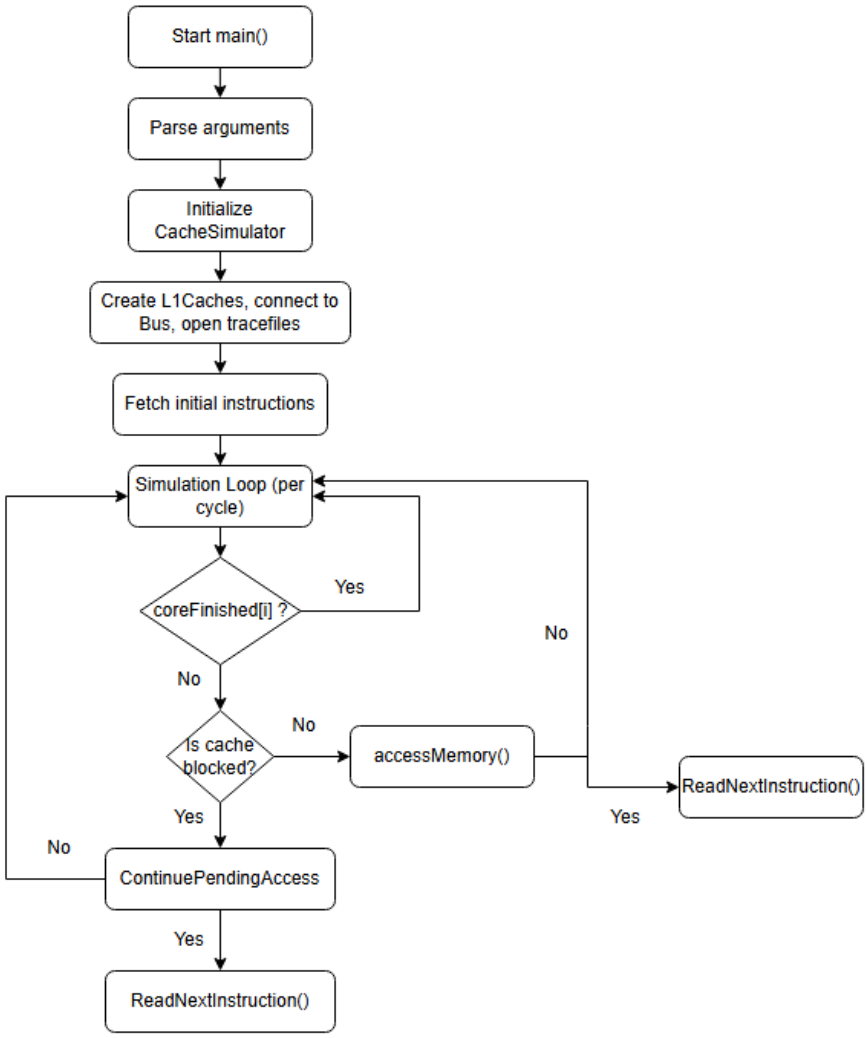

Figure 1: Simulation flowchart showing cache and bus logic

3 Assumptions

- When a core writes back to memory, its cache stays blocked.
- The bus can only process one transaction at any particular time. So issuing any of Bus_Rd, Bus_Rdx, Invalidate or Writeback would require the bus to be free at that time.
- If a core wants to issue a BusTransaction but the bus is busy at that time, then the transaction gets added to the pending_requests queue and the cache gets blocked. The queue follows the FIFO principle, as a result of which, running the simulation for any number of times on a particular input will provide the same results if the parameters s, E and b are kept constant.

- During a cache-to-cache transfer, suppose from Core1 to Core2 for example, then only Core2 stays blocked until the data transfer is completed. Core1 DOES NOT stall and continues processing its further instructions.
- Invalidation count of a core is assumed to be the number of times the Core sends an invalidate signal. We have assumed that in the case of a write hit in S state, the invalidation count will increase. It will also increase in the case of a write miss, even if none of the other cores have that address in M/E/S state.
- The execution cycles involve all the cycles when the core is performing an instruction. This also includes all the cycles that the core is waiting to receive data from the memory or from any other cache as well as all the cycles spent for writeback to memory.
- Idle cycles are those that the core spends waiting for the bus to become free so that it can issue its BusTransaction.

4 Graphical Analysis

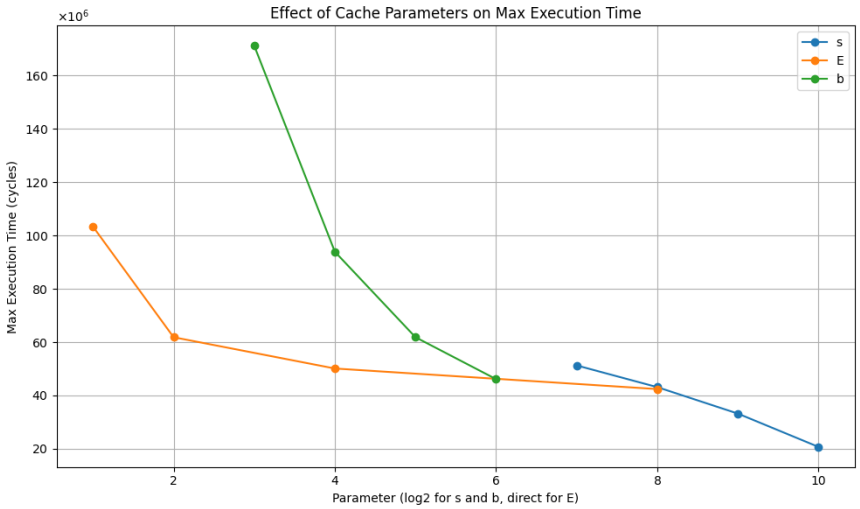

Figure 2: Maximum Execution Time vs Cache parameters

Observations:

- Increasing the **set index bits (s)** increases the number of cache sets, given by 2^s. As s increases, the execution time drops significantly because more sets reduce conflict misses, so cores are less likely to evict each other's data, improving hit rate and reducing bus traffic.
- Increasing **associativity (E)** allows more lines per set, which also helps reduce conflict misses. The drop in execution time is less dramatic than with s, and it flattens after a point because after moderate associativity (e.g., 4-way), additional gains are minimal because many conflict misses have already been eliminated.

- Increasing **block offset bits (b)** increases the block size: block size = 2^b bytes. The curve shows a very steep decrease in execution time with higher block sizes. This is because larger blocks bring in more spatial locality on each miss (one miss loads more useful data). Fewer total misses and fewer bus transactions.