Multi-Core Cache Simulation Report

April 30, 2025

Contents

1 Simulation Structure and Implementation

1.1 Flowcharts

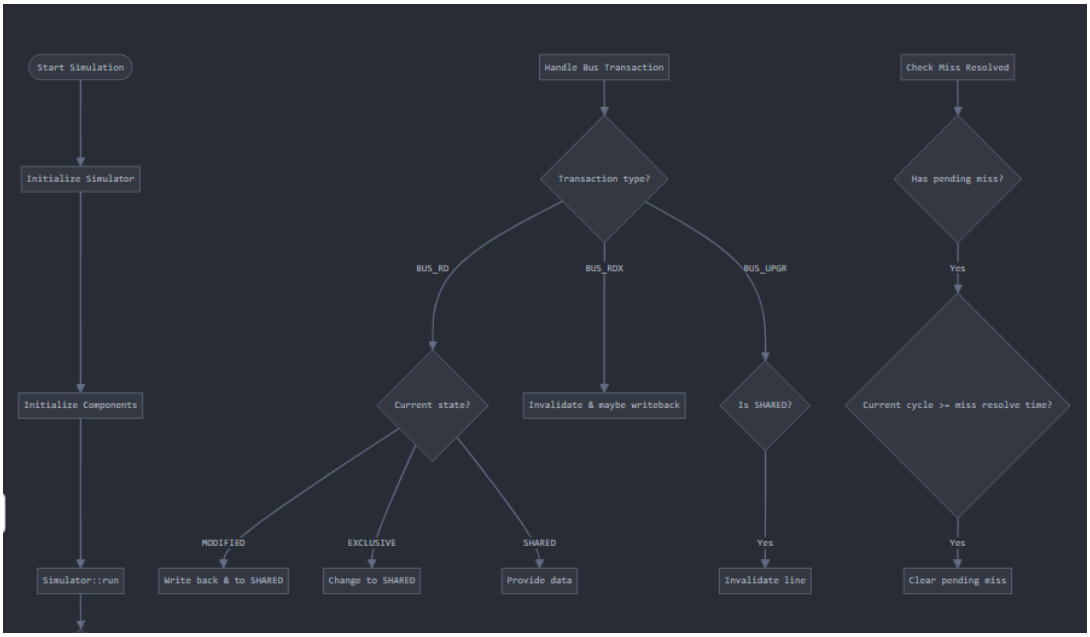

Figure 1: Flowchart depicting initialisation logic

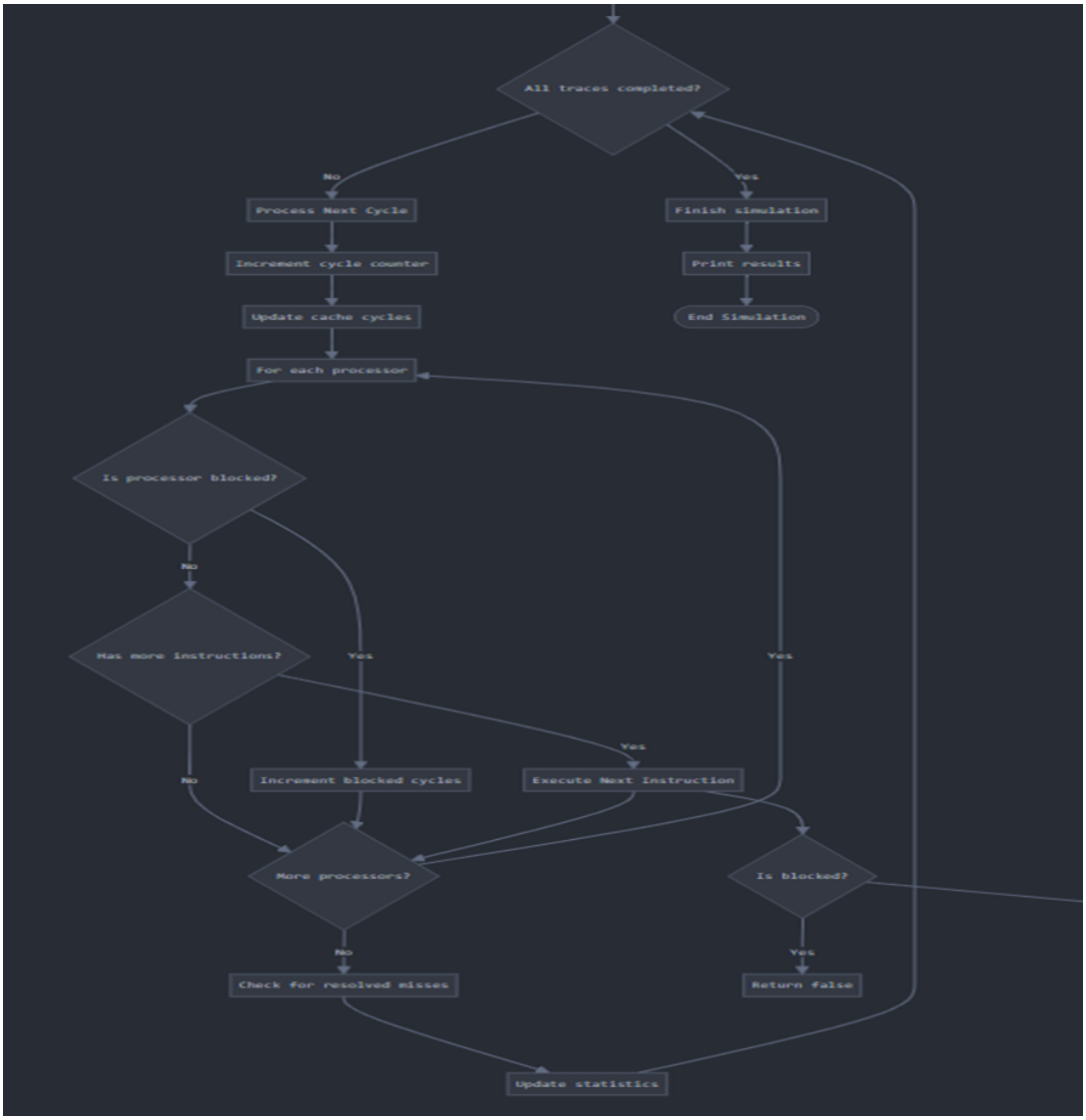

Figure 2: Flowchart depicting main loop logic

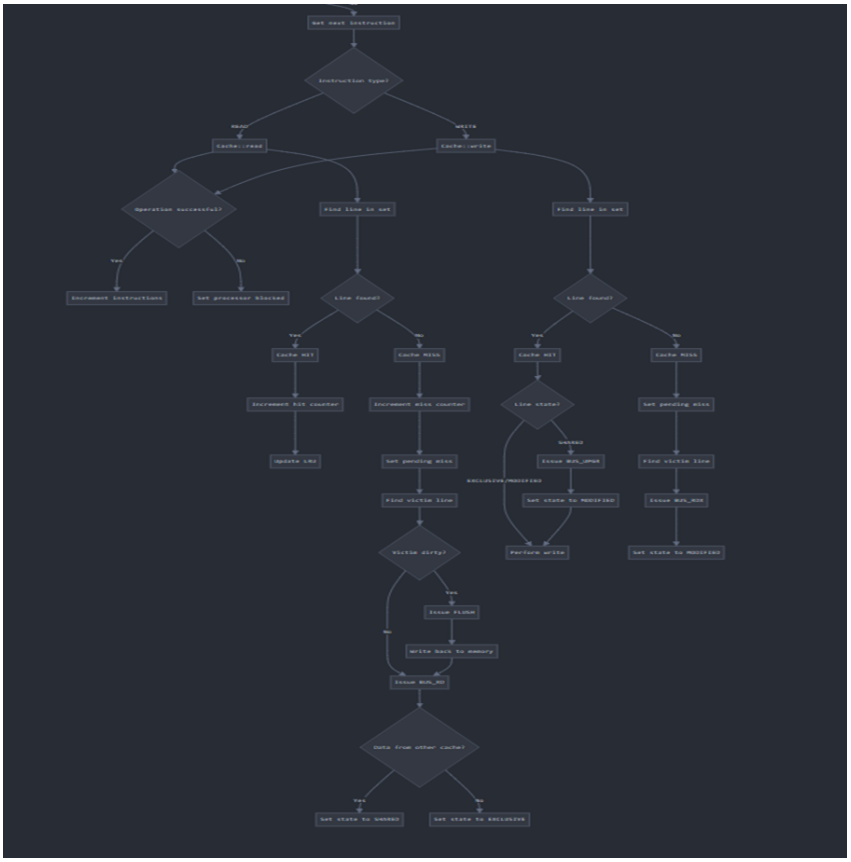

Figure 3: Still part of main loop; logic for blocks

1.2 Explanation of Classes and Functions

File: Simulator.h / Simulator.cpp

Class: Simulator

Main orchestrator for the entire multi-core simulation.

Data Members:

- SimulationConfig config: Configuration parameters
- TraceReader traceReader: Reads instruction traces
- MainMemory mainMemory: Shared memory system
- std::vector¡std::unique_ptr¡Cache¿¿ caches: L1 caches (one per core)
- std::vector¡std::unique_ptr¡Processor¿¿ processors: CPU cores
- std::ofstream logFile: Output log file
- unsigned int currentCycle: Current simulation cycle
- Various statistics counters (instructions, cycles, hits, misses, etc.)

Functions:

- Simulator(const SimulationConfig& config): Constructor, initializes simulation

- ~Simulator(): Destructor, closes log file
- bool initialize(): Opens trace files and initializes components
- void initializeComponents(): Sets up caches, processors, and coherence callbacks
- void run(): Main simulation loop
- bool processNextCycle(): Advances simulation by one cycle
- void logStatistics(): Logs current statistics
- unsigned int getTotalInstructions() const: Returns instruction count
- unsigned int getTotalCycles() const: Returns cycle count
- double getAverageMemoryAccessTime() const: Calculates average memory access time
- unsigned int getInvalidationCount() const: Returns total invalidations
- unsigned int getBusTrafficBytes() const: Returns total bus traffic
- void printResults() const: Outputs simulation results to console

File: Cache.h / Cache.cpp

Class: Cache

Implements an L1 cache with MESI coherence protocol.

Data Members:

- int coreId: Core ID this cache belongs to
- MainMemory& mainMemory: Reference to main memory
- int setBits, int blockBits: Address mapping configuration
- int numSets, int blockSize, int associativity: Cache configuration
- std::vector¡CacheSet¿ sets: Cache sets
- CoherenceCallback coherenceCallback: Function for issuing coherence requests
- Various counters (access, hit, miss, read, write, coherence)
- bool pendingMiss: Flag for tracking unresolved misses
- unsigned int missResolveTime: Cycle when miss will resolve
- unsigned int currentCycle: Current simulation cycle
- int dataSourceCache: ID of cache that provided data on a miss

Functions:

- Cache(int coreId, int numSets, int associativity, int blockSize, int setBits, int blockBits, MainMemory& mainMemory): Constructor

- bool read(const Address& addr): Handles read requests
- bool write(const Address& addr): Handles write requests
- bool checkMissResolved(): Checks if a pending miss has been resolved
- void setCycle(unsigned int cycle): Sets the current cycle
- void setCoherenceCallback(const CoherenceCallback& cb): Sets coherence callback
- void issueCoherenceRequest(...): Issues a bus transaction
- std::vector¡uint8_t¿ fetchBlockFromMemoryOrCache(...): Fetches a block
- bool handleBusTransaction(...): Processes bus transactions from other caches
- Various getters for statistics and configuration parameters
- const std::vector¡CacheSet¿& getSets() const: Access to cache sets
- bool hasPendingMiss() const: Returns if there's a pending miss
- unsigned int getEvictionCount() const: Returns eviction count
- unsigned int getWritebackCount() const: Returns writeback count

File: Processor.h / Processor.cpp (Provided in the question)

Class: Processor

Simulates a CPU core executing instructions.

Data Members:

- int coreId: Core ID
- TraceReader& traceReader: Reference to trace reader
- Cache& l1Cache: Reference to this core's L1 cache
- bool blocked: Whether processor is blocked on a cache miss
- unsigned int cyclesBlocked: Number of cycles spent blocked
- unsigned int instructionsExecuted: Number of instructions completed

Functions:

- Processor(int id, TraceReader& reader, Cache& cache): Constructor
- bool executeNextInstruction(): Attempts to execute the next instruction
- bool isBlocked() const: Returns if processor is blocked
- void setBlocked(bool state): Sets blocked state
- int getCoreId() const: Returns core ID
- bool hasMoreInstructions() const: Checks if there are more instructions

- unsigned int getCyclesBlocked() const: Returns cycles spent blocked
- unsigned int getInstructionsExecuted() const: Returns instructions executed
- void resetStats(): Resets processor statistics
- void incrementCyclesBlocked(): Increments blocked cycles counter

Other Important Classes (Referenced but not fully shown)

Class: CacheSet
Represents a set of cache lines.
Functions:

- CacheLine* findLine(uint32_t tag): Finds a line with matching tag
- CacheLine* findVictim(): Selects victim line for replacement
- void updateLRU(CacheLine* line): Updates LRU information
- const std::vector¡CacheLine¿& getLines() const: Access to cache lines

Class: CacheLine
Represents a single cache line.
Functions:

- bool isValid(): Checks if line is valid
- bool isDirty(): Checks if line has been modified
- uint32_t getTag(): Returns the tag
- MESIState getMESIState(): Returns MESI state
- void setMESIState(MESIState state): Sets MESI state
- const std::vector¡uint8_t¿& getData(): Access to data bytes
- void loadData(...): Loads data into line
- void writeWord(uint32_t offset, uint32_t data): Writes a word

Class: Address
Handles address decoding.
Functions:

- Address(uint32_t addr, int setBits, int blockBits): Constructor
- uint32_t getTag(): Returns address tag
- uint32_t getIndex(): Returns set index
- uint32_t getOffset(): Returns block offset
- uint32_t getBlockAddress(): Returns block-aligned address

Class: MainMemory
Simulates main memory.
Functions:

- std::vector¡uint8_t¿ readBlock(uint32_t blockAddr): Reads a block
- void writeBlock(uint32_t blockAddr, const std::vector¡uint8_t¿& data): Writes a block

Class: TraceReader
Reads instruction traces.
Functions:

- bool openTraceFiles(): Opens trace files
- bool hasMoreInstructions(int coreId): Checks for more instructions
- Instruction getNextInstruction(int coreId): Gets next instruction
- bool allTracesCompleted(): Checks if all traces are done

Enum: MESIState
Represents the states in MESI protocol:

- MODIFIED: Exclusive ownership with modified data
- EXCLUSIVE: Exclusive ownership with clean data
- SHARED: Multiple caches have clean copies
- INVALID: Line not present or invalidated

Enum: BusTransaction
Types of bus transactions:

- BUS_RD: Read request
- BUS_RDX: Exclusive read request
- BUS_UPGR: Upgrade request (Shared to Modified)
- INVALIDATE: Explicit invalidation request
- FLUSH: Writeback notification

Struct: SimulationConfig
Configuration parameters for the simulation:

- Application name
- Cache parameters (set bits, block bits, associativity)
- Output file

Struct: Instruction
Represents a single processor instruction:

- Address
- Type (READ or WRITE)
- Validity flag

Global Variables

- std::vector¡Cache*¿ allCaches: Global vector of cache pointers for coherence
- static unsigned int busBusyUntil: Global bus-reservation timestamp

2 Simulation Determinism

Because the algorithm is deterministic, the output parameters will not change on different runs.

3 Performance Analysis

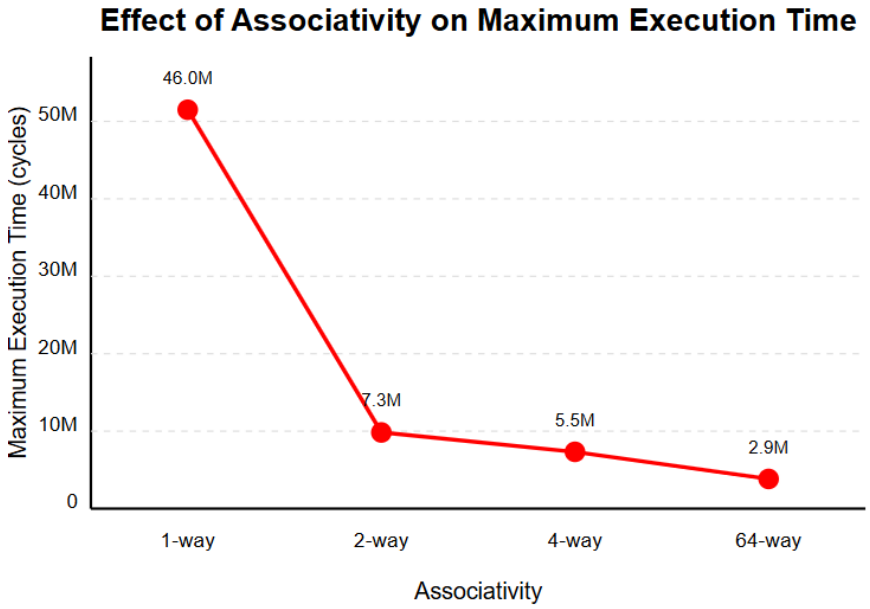

Figure 4: Performance vs. Parameter

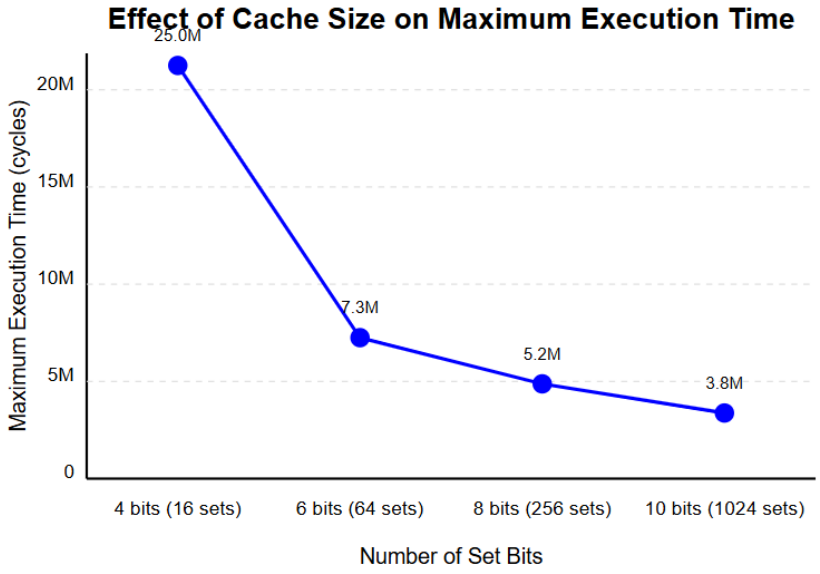

Figure 5: Another performance metric

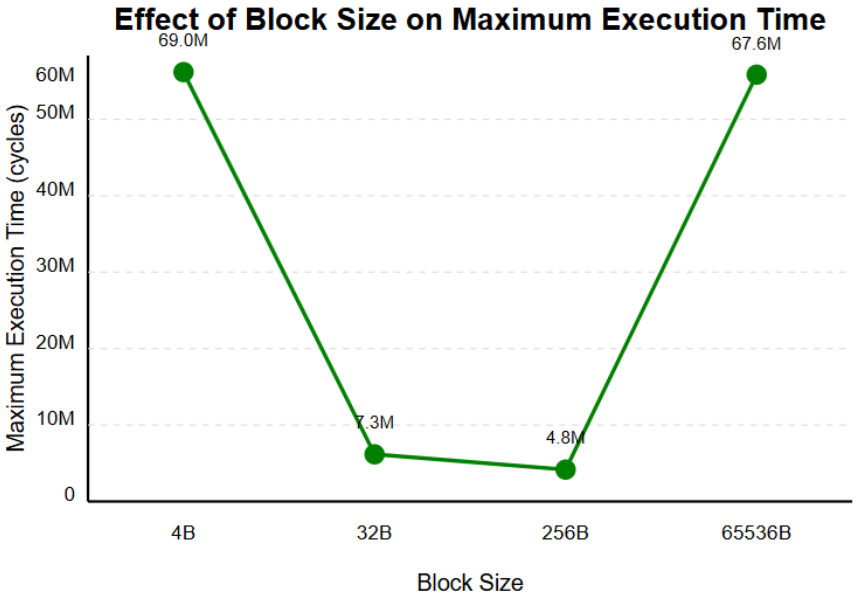

Figure 6: Block size

3.1 Effect of Cache Size (Number of Set Bits)

Set Bits	Number of Sets	Max Execution Time	Improvement vs Baseline
4	16	25,013,798	-242.3% (worse)
6	64	7,307,218	Baseline
8	256	5,243,058	+28.2%
10	1024	3,797,270	+48.0%

Figure 7: Effect of Cache Size

Analysis:

- Increasing the cache size has a significant positive impact on performance
- Going from 16 sets to 64 sets (4 to 6 bits) reduces execution time by 70.8%
- Additional increases show diminishing returns, with 1024 sets providing only 48% improvement over 64 sets

Explanation: Having a greater number of set bits decreases the execution time (up to the point recorded) because a greater number of blocks are stored in the cache at any time, decreasing the miss rate and thus avoiding the greater time taken to access from memory. There is a significant improvement from 4 to 6 set bits, because extremely low (4) bits in the cache lead to a very large portion of accesses being misses and frequent eviction of data.

2. Effect of Associativity

Associativity	Max Execution Time	Improvement vs Baseline
1-way	45,993,498	-529.4% (worse)
2-way	7,307,218	Baseline
4-way	5,537,346	+24.2%
64-way	2,910,918	+60.2%

Figure 8: Effect of Associativity

Analysis:

- Direct-mapped caches (1-way) perform very poorly due to conflict misses
- The jump from direct-mapped to 2-way associative yields a massive 84.1% reduction in execution time

- Higher associativity continues to improve performance, with 64-way showing the best results
- This suggests the workload has significant conflict misses that benefit from higher associativity

Explanation: We find the associativity increase also keeps on decreasing execution time, although the decrease is very minimal at higher associativity. A greater associativity decreases the execution time by decreasing the number of evictions; a block occupying the same address (set bits in the cache) need not evict another block in the cache if it is occupied, and for a fully associative (64 size) set, a block can be placed anywhere in the cache, making the need for evictions very minimal.

3. Effect of Block Size

Block Size (bits)	Block Size (bytes)	Max Execution Time	Improvement vs Baseline
2	4	68,984,298	-843.9% (worse)
5	32	7,307,218	Baseline
8	256	4,763,478	+34.8%
16	65536	67,615,844	-825.3% (worse)

Figure 9: Effect of Block Size

Analysis:

- Block size shows a U-shaped curve in performance impact
- Very small blocks (4 bytes) perform extremely poorly
- Very large blocks (64KB) perform almost as poorly as very small blocks
- The optimal range appears to be between 32 bytes and 256 bytes
- Small blocks (4B) fail to exploit spatial locality, causing excessive misses
- Very large blocks (64KB) waste bandwidth and cause excessive evictions/conflicts

Explanation: A U shaped curve is expected when varying block size. Initially, increases in the block size are beneficial, as they utilise spatial locality more (a greater range of words around an accessed word are accessible when a block is brought into the cache) which means, as per spatial locality when nearby words are accessed there is lesser likelihood of misses. However, after a point increases lead to decrease in execution time because increasing block size also increases the miss penalty (when misses do occur, a larger number of bytes need to be transferred back into memory) and also increases competition for blocks (a greater block size means the cache can accommodate fewer blocks, leading to more misses).

4. Interesting Trace Analysis - Demonstrating False Sharing

Each file contains a single memory write to a different offset:

```
app4_proc0.trace:  W 0x2000
app4_proc1.trace:  W 0x2008
app4_proc2.trace:  W 0x2010
app4_proc3.trace:  W 0x2018
```

Each address is 8 bytes apart (0x8), so:

- All four addresses lie within a single 64-byte cache line (assuming block size b = 6, i.e., 64 bytes).
- The first address is 0x2000, and the last is 0x2018, so the entire line spans from 0x2000 to 0x203F.

This cause false sharing because all processors (0–3) are writing to different variables that share the same cache line:

- Suppose each processor has its own private L1 cache (a common setup).
- When proc0 writes to 0x2000, it loads the cache line [0x2000–0x203F] into its cache.
- Then proc1 writes to 0x2008 — that line must be loaded into proc1's cache and invalidated in proc0's.
- Then proc2 writes to 0x2010, evicting from proc1's cache.
- Then proc3 writes to 0x2018, invalidating from proc2's.

This ping-ponging means:

- Every write forces the line to migrate between cores
- Even though each core touches only its own 8-byte slice, the full cache line must move

So each write results in a cache miss due to coherence-induced invalidation.