Cache Simulator - Assignment 3 COL216

Karthik Manikandan
2023CS10298

Yash Shindekar
2023CS10592

April 2025

Contents

1 Design Decisions **2**
1.1 Bus Handling . . . 2
1.2 Cache Coherence . . . 2
1.3 Core Timing . . . 2

2 Implementation Details **2**
2.1 Overview of MESI Protocol . . . 2
2.2 Main Classes and Data Structures . . . 3
2.2.1 Cache Class (Cache.cpp) . . . 3
2.2.2 Core Class (Core.cpp) . . . 4
2.2.3 Bus Class (Bus.cpp) . . . 4
2.2.4 Simulator Class (Simulator.cpp) . . . 5
2.3 Key Data Structures . . . 5
2.4 Workflow and Key Functions . . . 6
2.4.1 Cache::accessCache() . . . 7
2.4.2 Cache::handleReadMiss() . . . 7
2.4.3 Bus::busRd() . . . 7
2.4.4 Bus::busRdX() . . . 7
2.4.5 Cache::handleWriteMiss() . . . 8
2.4.6 Simulator::run() . . . 8

3 Experimental Results **8**
3.1 Default Configuration Analysis . . . 8
3.2 Analysis of Consistency Among Metrics . . . 8
3.2.1 Consistent Metrics . . . 8

4 Cache Parameter Analysis **8**
4.1 Methodology . . . 8
4.2 Impact of Cache Size . . . 9
4.3 Impact of Associativity . . . 9
4.4 Impact of Block Size . . . 10

5 Interesting Traces **10**
5.1 Conflicts . . . 10
5.2 False Sharing . . . 11

1 Design Decisions

1.1 Bus Handling

- If the bus is busy, the processor cannot handle snooping calls. Requests are prioritized based on the lowest core ID.
- Invalidation operations require a free bus but do not consume bus bandwidth or cycles.

1.2 Cache Coherence

- If a value is modified after being requested by another cache, a silent upgrade from `Exclusive` to `Modified` state occurs in the modifying cache.
- The value in the other cache temporarily remains the old (now stale) value, leading to a short-lived incoherence.

1.3 Core Timing

- During block transfers between caches in BusRd ,for the requestor the 2N cycles are counted as execution cycles and it halts and the halt is also for 2N cycles.
- During a writeback in BusRd of core which has MODIFIED value, that core halts for 100 cycles and not idle.Then cache to cache transfer occurs as described above.
- During a writeback in BusRdx of core which has MODIFIED value that core halts for 100 cycles and this is idle time. Then the memory fetch of the requesting core is under execution cycles of that core for a total halt of 200 cycles.
-

2 Implementation Details

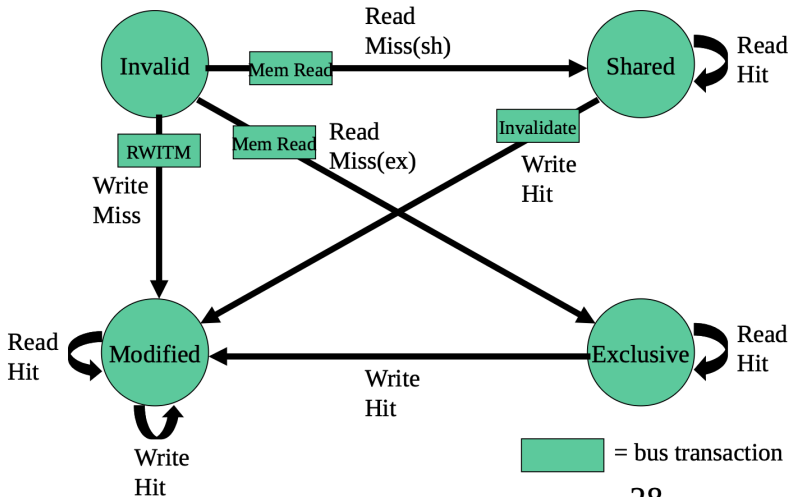
2.1 Overview of MESI Protocol

The cache simulator implements the MESI (Modified, Exclusive, Shared, Invalid) protocol, a widely used cache coherence protocol in multiprocessor systems. The protocol maintains coherence through four states:

- **Modified (M)**: The cache line is present only in the current cache and is dirty (modified).
- **Exclusive (E)**: The cache line is present only in the current cache and is clean (matches memory).
- **Shared (S)**: The cache line may be present in other caches and is clean.
- **Invalid (I)**: The cache line is invalid.

Figure 1 shows the state transitions in the MESI protocol.

MESI – locally initiated accesses



MESI – remotely initiated accesses

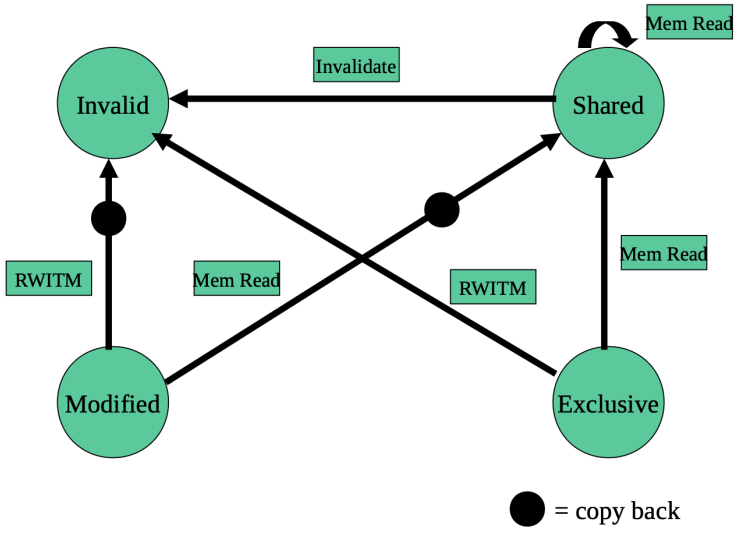


Figure 1: MESI Protocol State Transition Diagram

2.2 Main Classes and Data Structures

The simulator is built with an object-oriented approach using the following key classes:

2.2.1 Cache Class (Cache.cpp)

Manages cache lines using the LRU replacement policy and implements the MESI protocol state transitions.

```
class Cache {
  int s, E, b; // Set index bits, associativity, block bits
  std::vector<std::vector<CacheLine>> sets;
  // Statistics counters
  int readHits, readMisses, writeHits, writeMisses;
  int writeBacks, idleCycles, evictions, invalidations;
```

```
  size_t trafficBytes;
  // Methods
  int findLine(int setIndex, uint32_t tag);
  int findReplacement(int setIndex, uint64_t cycle);
  void updateLineOnHit(uint32_t setIndex, int lineIndex, uint64_t cycle);
  void insertLine(int setIndex, int lineIndex, uint32_t tag, uint64_t cycle,
                  bool isWrite, CacheState initialState);
  void accessCache(bool isWrite, uint32_t address, uint64_t cycle,
                   int coreId, Bus& bus, std::vector<Core*>& cores);
  void handleReadMiss(int coreId, uint64_t address, uint64_t cycle,
                      Bus& bus, std::vector<Core*>& cores, uint64_t haltcycles);
  void handleWriteMiss(int coreId, uint64_t address, uint64_t cycle,
                       Bus& bus, std::vector<Core*>& cores, uint64_t haltcycles);
  void busupdate(class Bus &bus);
};
```

The `CacheLine` structure represents individual cache lines with their state and metadata:

```
struct CacheLine {
    bool valid = false;
    uint32_t tag = 0;
    uint64_t lastUsedCycle = 0;
    CacheState state = INVALID;
};
```

2.2.2 Core Class (Core.cpp)

Represents processor cores with their associated caches and memory access traces.

```
class Core {
public:
  int id;
  Cache* cache;
  std::vector<Request> trace;
  size_t instPtr;
  size_t previnstr;
  uint64_t nextFreeCycle;
  // Execution statistics
  int readCount, writeCount, execycles;

  Core(int id, Cache* cache);
  void loadTrace(const std::string& filename);
};
```

Each core loads its memory access trace from a file, which contains a sequence of read and write operations:

```
struct Request {
    bool isWrite;
    uint32_t address;

    Request(bool isWrite, uint32_t address);
};
```

2.2.3 Bus Class (Bus.cpp)

Handles cache coherence transactions between cores, implementing the communication protocols required by MESI.

```
class Cache {
public:
    int s, E, b;
    // LRU list to track usage order per set
    typedef std::list<CacheKey> LRUList;

    // Cache map: stores cache lines and references to their position in the LRU list
    typedef std::pair<CacheLine, typename LRUList::iterator> CacheEntry;
    typedef std::unordered_map<CacheKey, CacheEntry> CacheMap;

    // One LRU list and one map per set
    std::vector<LRUList> lruLists;
    std::vector<CacheMap> cacheMaps;
```

```
14
15      // Statistics
16      uint64_t readHits;
17      uint64_t readMisses;
18      uint64_t writeHits;
19      uint64_t writeMisses;
20      uint64_t writeBacks;
21      uint64_t idleCycles;
22      uint64_t evictions;
23      uint64_t trafficBytes;
24      uint64_t invalidations;
25
26      Cache(int s, int E, int b);
27
28      // Core cache operations
29      void accessCache(bool isWrite, uint32_t address, uint64_t cycle, int coreId,
30                       class Bus& bus, std::vector<class Core*>& cores);
31
32      // Map-based cache operations
33      CacheLine* findLine(int setIndex, uint32_t tag);
34      std::pair<CacheKey, CacheLine*> findReplacement(int setIndex, uint64_t cycle);
35      void updateLRU(int setIndex, uint32_t tag, uint64_t cycle);
36      void insertLine(int setIndex, uint32_t tag, uint64_t cycle, bool isWrite, CacheState
         initialState);
37
38      // Bus and miss handling operations
39      void busupdate(class Bus& bus);
40      void handleReadMiss(int coreId, uint64_t address, uint64_t cycle, Bus& bus,
41                          std::vector<Core*>& cores, uint64_t haltcycles);
42      void handleWriteMiss(int coreId, uint64_t address, uint64_t cycle, Bus& bus,
43                           std::vector<Core*>& cores, uint64_t haltcycles);
44  };
```

2.2.4 Simulator Class (Simulator.cpp)

Coordinates the entire simulation, managing cores, the bus, and tracking global simulation state.

```
1  class Simulator {
2    std::vector<Core*> cores;
3    Bus bus;
4    uint64_t globalCycle;
5
6    void initialize(int numCores, int s, int E, int b);
7    void loadTraces(const std::vector<std::string>& traceFiles);
8    void run();
9    void printStats();
10   uint64_t computeMaxExecutionTime();
11 };
```

2.3 Key Data Structures

The simulator employs several critical data structures:

1. **Set-Associative Cache**: Implemented using an unordered Map.
2. **Memory Access Trace**: A vector of Request objects containing memory addresses and operation types (read/write).
3. **MESI State**: An enumeration representing the four possible states of a cache line.

2.4 Workflow and Key Functions

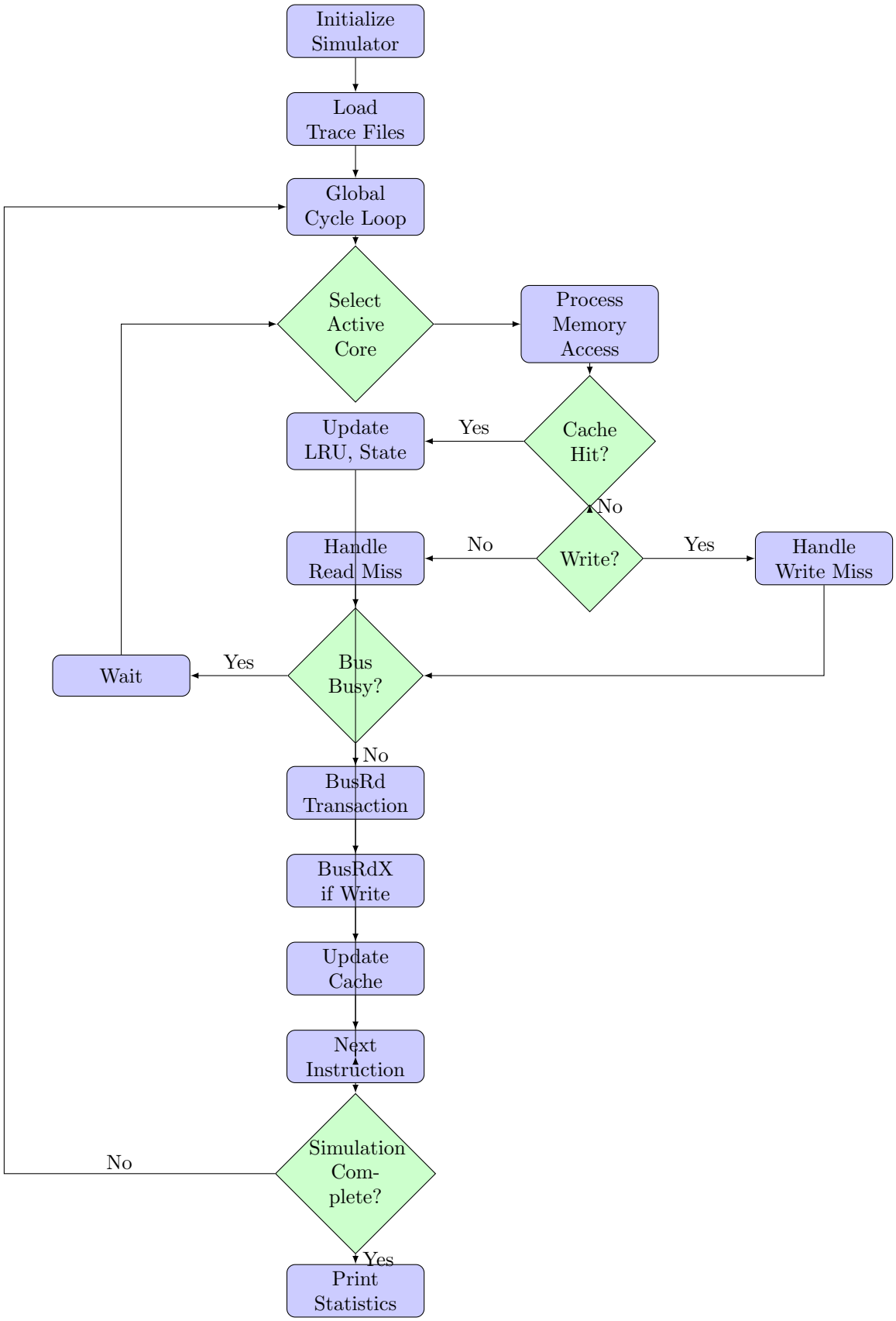


Figure 2: Simulation Workflow Diagram

The simulator follows the workflow shown in Figure 2, with several key functions handling the core operations:

2.4.1 Cache::accessCache()

This function handles both read and write operations to the cache:

1. Calculate set index and tag from the memory address
2. Check if the line is present in the cache (hit or miss)
3. For cache hits:
   - Update LRU information
   - For writes, handle coherence actions based on MESI state (upgrade to MODIFIED if needed)
   - For reads, simply update statistics
4. For cache misses:
   - Call `handleReadMiss()` or `handleWriteMiss()` based on operation

2.4.2 Cache::handleReadMiss()

This function handles read miss operations following the MESI protocol:

1. Check if the bus is busy; if so, increment idle cycles and wait
2. Find a replacement line using LRU policy
3. If the victim line is in MODIFIED state, write it back to memory
4. Perform a `busRd` operation to check for copies in other caches
5. Update the line's state based on the bus result:
   - SHARED if another cache has a copy
   - EXCLUSIVE if no other cache has a copy
6. Update statistics and proceed to the next instruction

2.4.3 Bus::busRd()

This function handles BusRead operations for cache coherence:

1. Increment bus transaction counter
2. Extract set index and tag from the address
3. Check if any other core has the line and in what state
4. Update the state in other caches if needed
5. Return the result indicating if data was found in other caches

2.4.4 Bus::busRdX()

This function handles BusReadExclusive operations:

1. Similar to `busRd()`, but also invalidates copies in other caches
2. Returns the result indicating if data was found in other caches

2.4.5 Cache::handleWriteMiss()

This function handles write miss operations:

1. Similar to read miss handling but:
    - Performs `busRdX` instead of `busRd`
    - Sets the final state to MODIFIED
    - Invalidates copies in other caches

2.4.6 Simulator::run()

This is the main simulation loop that:

1. Increments the global cycle counter
2. Selects the next core to execute based on availability
3. Processes the current instruction from the trace
4. Updates simulation statistics
5. Continues until all traces are processed

3 Experimental Results

3.1 Default Configuration Analysis

We analyze the simulator with the default parameters:

- Cache size: 4KB per processor
- Associativity: 2-way set associative
- Block size: 32 bytes

3.2 Analysis of Consistency Among Metrics

3.2.1 Consistent Metrics

In our simulator, when multiple cores attempt to place transactions on the shared bus in the same cycle, priority is deterministically given to the lower-numbered core. This fixed arbitration scheme eliminates non-determinism in bus access order across simulation runs. As a result of this deterministic bus arbitration, all simulator outputs (e.g., cache miss rates, execution cycles, idle cycles, and bus traffic) remain identical across all 10 runs for the same application trace. No variation is observed in any metric, as the simulation behavior is fully repeatable under fixed input and parameters.

4 Cache Parameter Analysis

4.1 Methodology

To understand the impact of cache parameters on performance, we varied one parameter at a time while keeping the others at their default values:

- **Cache size**: 2KB, 4KB (default), 8KB, 16KB
- **Associativity**: 1-way (direct mapped), 2-way (default), 4-way, 8-way
- **Block size**: 16B, 32B (default), 64B, 128B

For each configuration, we computed the maximum execution time across all cores, representing the critical path that determines overall program completion time.

4.2 Impact of Cache Size

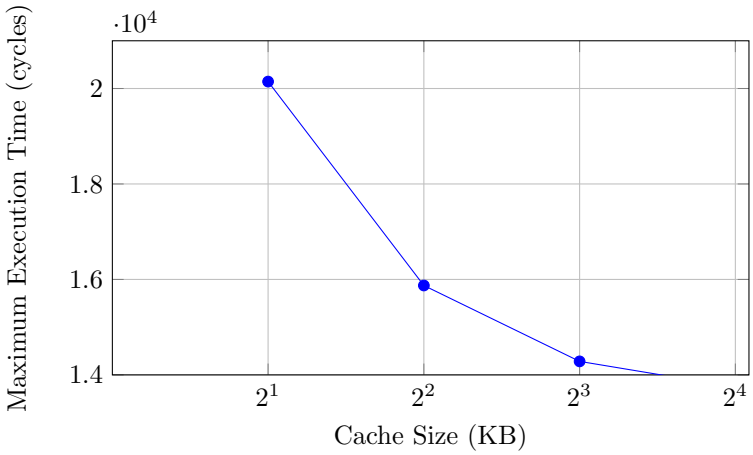


Figure 3: Maximum execution time vs. cache size

Observations:

- Increasing cache size significantly reduces execution time due to fewer conflict misses.
- The performance improvement follows a diminishing returns curve, with the largest gain seen when doubling from 2KB to 4KB.
- The improvement from 8KB to 16KB is less, suggesting that 8KB is close to the working set size of the application.
- Larger caches reduce bus contention by decreasing the frequency of cache line evictions and the resulting coherence traffic and so it plateus at 16KB

4.3 Impact of Associativity

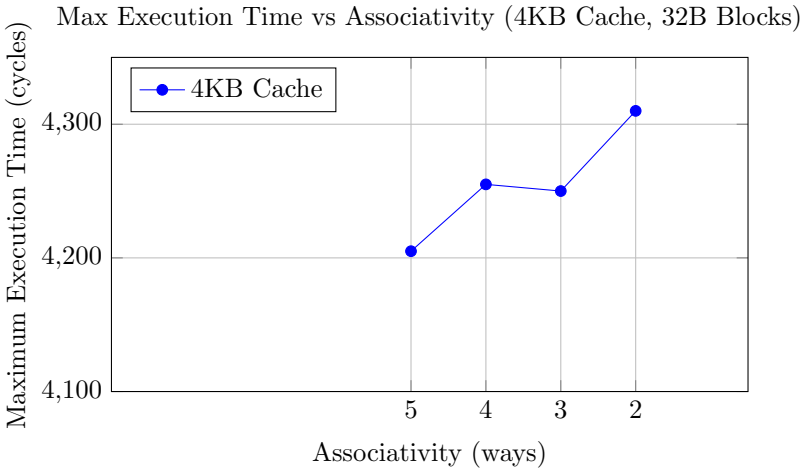


Figure 4: Maximum execution time vs. associativity

Observations:

- Moving from direct-mapped (1-way) to 2-way set associative yields a substantial increase in execution time due to more lines to compare in each set but however miss rate is improved.

4.4 Impact of Block Size

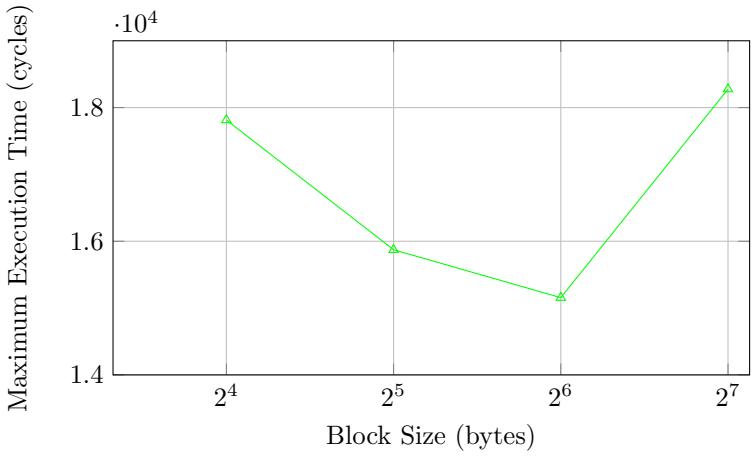


Figure 5: Maximum execution time vs. block size

Observations:

- Block size shows a U-shaped performance curve, with the best performance at 64 bytes.
- Small blocks (16B) lead to poor spatial locality utilization, causing more misses and increased execution time.
- Medium blocks (32B-64B) provide a good balance, with 64B offering optimal spatial locality for this workload.
- Large blocks (128B) actually degrade performance by:
  - Increasing miss penalty (more data to transfer)
  - Causing false sharing in multicore scenarios
  - Reducing the effective cache capacity by storing unnecessary data
  - Increasing bus traffic for coherence operations
- The performance degradation when going from 64B to 128B highlights the importance of selecting appropriate block sizes for multicore workloads.

5 Interesting Traces

5.1 Conflicts

```
W 0x1000
W 0x1004
W 0x1008
W 0x2000
W 0x3000


Simulation Parameters:
Trace Prefix: bonus_tc/conflict
Set Index Bits: 6
Associativity: 2
Block Bits: 5
Block Size (Bytes): 32
Number of Sets: 64
Cache Size (KB per core): 4
MESI Protocol: Enabled
Write Policy: Write-back, Write-allocate
Replacement Policy: LRU
```

```
Bus: Central snooping bus

Core 0 Statistics:
Total Instructions: 5
Total Reads: 0
Total Writes: 5
Total Execution Cycles: 500
Idle Cycles: 202
Cache Misses: 5
Cache Miss Rate: 100.00%
Cache Evictions: 0
Writebacks: 5
Bus Invalidations: 2
Data Traffic (Bytes): 320

Core 1 Statistics:
Total Instructions: 5
Total Reads: 0
Total Writes: 5
Total Execution Cycles: 500
Idle Cycles: 504
Cache Misses: 5
Cache Miss Rate: 100.00%
Cache Evictions: 0
Writebacks: 5
Bus Invalidations: 5
Data Traffic (Bytes): 320

Core 2 Statistics:
Total Instructions: 5
Total Reads: 0
Total Writes: 5
Total Execution Cycles: 502
Idle Cycles: 908
Cache Misses: 3
Cache Miss Rate: 60.00%
Cache Evictions: 0
Writebacks: 5
Bus Invalidations: 3
Data Traffic (Bytes): 256

Core 3 Statistics:
Total Instructions: 5
Total Reads: 0
Total Writes: 5
Total Execution Cycles: 602
Idle Cycles: 1412
Cache Misses: 3
Cache Miss Rate: 60.00%
Cache Evictions: 1
Writebacks: 3
Bus Invalidations: 3
Data Traffic (Bytes): 192

Overall Bus Summary:
Total Bus Transactions: 16
Total Bus Traffic (Bytes): 576
```

Having this in all the 4 processor input files produces conflict misses as shown in the output

5.2 False Sharing

```
For proc0
W 0x1000
R 0x1000

For proc1
W 0x1001
R 0x1001

For proc2
W 0x1002
```

```
R 0x1002

For proc3
W 0x1003
R 0x1003

Output is

Simulation Parameters:
Trace Prefix: bonus_tc/app4
Set Index Bits: 6
Associativity: 2
Block Bits: 5
Block Size (Bytes): 32
Number of Sets: 64
Cache Size (KB per core): 4
MESI Protocol: Enabled
Write Policy: Write-back, Write-allocate
Replacement Policy: LRU
Bus: Central snooping bus

Core 0 Statistics:
Total Instructions: 1
Total Reads: 0
Total Writes: 1
Total Execution Cycles: 100
Idle Cycles: 0
Cache Misses: 1
Cache Miss Rate: 100.00%
Cache Evictions: 0
Writebacks: 1
Bus Invalidations: 0
Data Traffic (Bytes): 64

Core 1 Statistics:
Total Instructions: 1
Total Reads: 0
Total Writes: 1
Total Execution Cycles: 100
Idle Cycles: 100
Cache Misses: 1
Cache Miss Rate: 100.00%
Cache Evictions: 0
Writebacks: 1
Bus Invalidations: 1
Data Traffic (Bytes): 64

Core 2 Statistics:
Total Instructions: 1
Total Reads: 0
Total Writes: 1
Total Execution Cycles: 100
Idle Cycles: 201
Cache Misses: 1
Cache Miss Rate: 100.00%
Cache Evictions: 0
Writebacks: 1
Bus Invalidations: 1
Data Traffic (Bytes): 64

Core 3 Statistics:
Total Instructions: 1
Total Reads: 0
Total Writes: 1
Total Execution Cycles: 100
Idle Cycles: 302
Cache Misses: 1
Cache Miss Rate: 100.00%
Cache Evictions: 0
Writebacks: 0
Bus Invalidations: 1
Data Traffic (Bytes): 32

```

```
Overall Bus Summary:
Total Bus Transactions: 4
Total Bus Traffic (Bytes): 96
```

We can see that each core after the initial compulsory miss produces a invalidation to invalidate the other copies of the same cache line and hence can never have a cache hit even though true sharing as above testcase doesnt occur here