Cache Coherence Simulator Implementation Report

Cache Simulator

April 30, 2025

Source Code Repository

The complete source code for this project is available at:

`https://github.com/perle01/cache`

1 Implementation Details

This section describes the implementation of our cache coherence simulator, which models a multiprocessor system with private L1 caches connected via a shared bus implementing the MESI (Modified-Exclusive-Shared-Invalid) cache coherence protocol.

1.1 System Architecture

The simulator implements a symmetric multiprocessor system with the following characteristics:

- Multiple processor cores, each with a private L1 cache
- A central shared bus connecting all caches
- MESI cache coherence protocol
- Write-back, write-allocate caching policy
- LRU (Least Recently Used) replacement policy

Figure 1 illustrates the overall architecture of the simulated system.

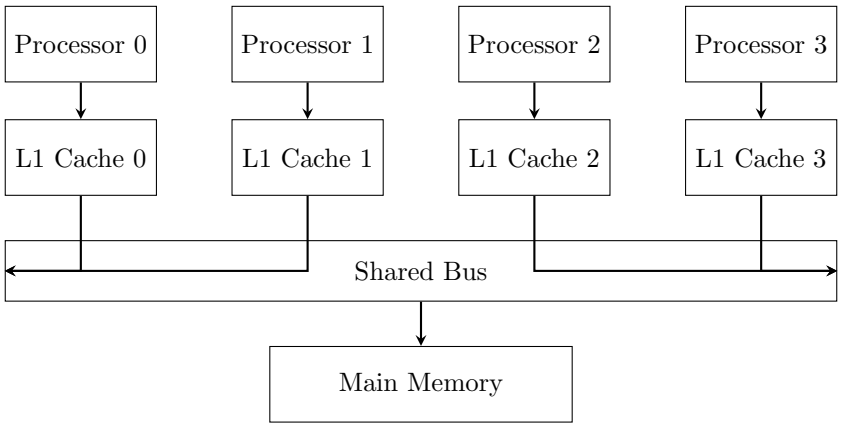


Figure 1: System Architecture with 4 Processors

1.2 Main Classes and Data Structures

The implementation consists of several key classes that work together to simulate the cache coherence protocol. Figure 2 shows the main classes and their relationships.

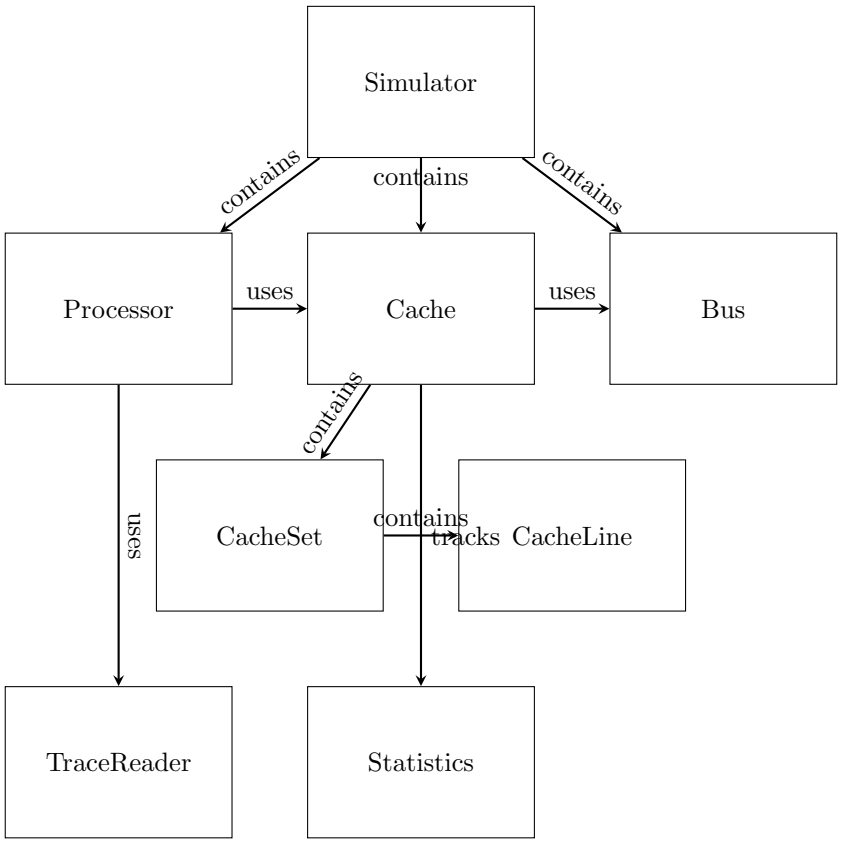


Figure 2: Class Diagram of the Simulator

The key classes and their roles are:

1.2.1 Simulator

The central coordinator class that initializes and runs the simulation. It:

- Creates and connects processors, caches, and the bus
- Runs the simulation cycle by cycle
- Collects and reports statistics
- Detects and resolves deadlocks

Key data structures:

- `std::vector<std::shared_ptr<Processor>>` - Collection of processor cores
- `std::vector<std::shared_ptr<Cache>>` - Collection of L1 caches

- `std::shared_ptr<Bus>` - The shared bus

1.2.2 Processor

Models a CPU core that executes memory operations from a trace file:

- Reads memory operations (read/write) from trace files
- Issues memory operations to its L1 cache
- Tracks execution time and statistics

Key data structures:

- `std::queue<MemoryReference>` - Queue of pending memory operations
- `std::shared_ptr<Cache>` - Reference to the associated L1 cache

1.2.3 Cache

Implements a set-associative cache with MESI protocol support:

- Handles read and write requests from the processor
- Implements cache coherence actions via the bus
- Manages cache lines and implements replacement policy
- Responds to bus snooping operations

Key data structures:

- `std::vector<CacheSet>` - Array of cache sets
- `Statistics` - Tracks cache performance metrics

1.2.4 CacheSet

Represents a set in the cache:

- Contains multiple cache lines based on associativity
- Implements LRU (Least Recently Used) replacement policy

Key data structures:

- `std::vector<CacheLine>` - Cache lines in the set
- `std::vector<int>` - LRU counters for replacement policy

1.2.5 CacheLine

Represents a single cache line:

- Stores data, tag, and coherence state (MESI)
- Provides methods to read and write data

Key data members:

- `uint32_t tag` - Tag bits of the address
- `CacheState state` - MESI protocol state (Modified, Exclusive, Shared, Invalid)
- `uint8_t* data` - Actual data stored in the cache line

1.2.6 Bus

Implements the shared bus connecting all caches:

- Facilitates communication between caches
- Implements bus transactions for cache coherence
- Handles snooping operations

Key data structures:

- `std::vector<Cache*>` - References to all caches
- `std::queue<BusTransaction>` - Queue of pending bus transactions

1.2.7 Statistics

Collects and reports performance metrics:

- Tracks cache hits, misses, evictions, writebacks
- Counts bus operations and traffic
- Measures execution time and other performance metrics

1.2.8 TraceReader

Reads memory access traces from files:

- Parses trace file format
- Provides memory access operations to processors

1.3 MESI Protocol Implementation

The MESI protocol is implemented through the cooperation of `Cache` and `Bus` classes. Each cache line has one of four possible states:

- **M**odified: Line is valid, owned exclusively, and has been modified (dirty)
- **E**xclusive: Line is valid, owned exclusively, and matches memory (clean)
- **S**hared: Line is valid, may exist in other caches, and matches memory
- **I**nvalid: Line does not contain valid data

The state transitions are triggered by processor requests and bus snooping operations as shown in Figure **??**.

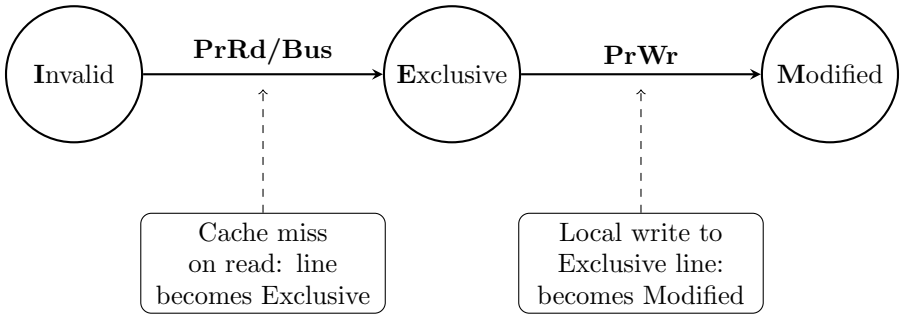


Figure 3: MESI Protocol: Read Miss and Local Write Transitions

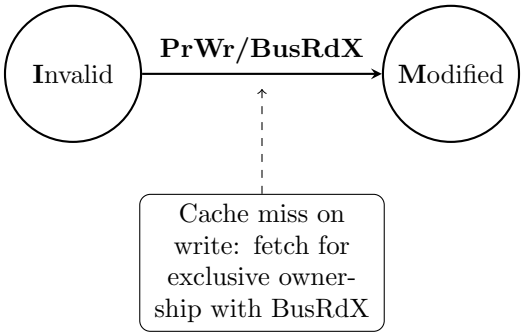


Figure 4: MESI Protocol: Write Miss Transition

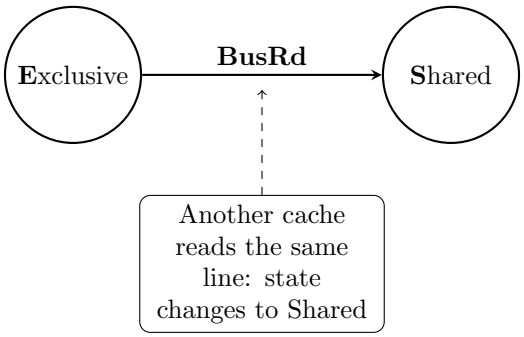


Figure 5: MESI Protocol: Transition to Shared State on Bus Read

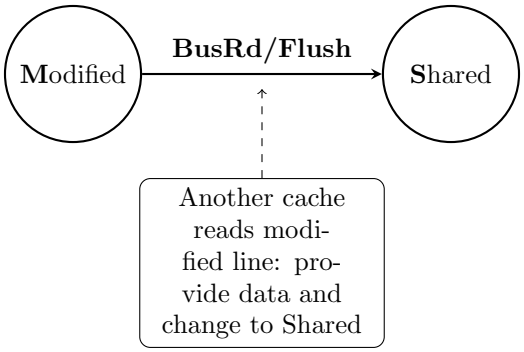


Figure 6: MESI Protocol: Sharing Modified Data

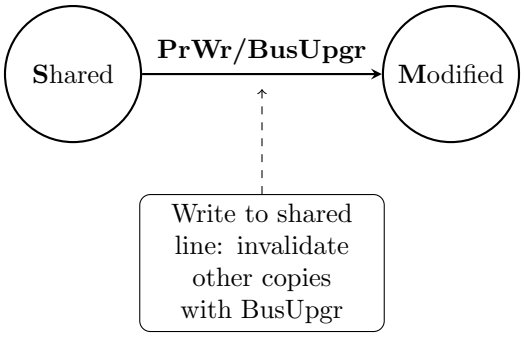


Figure 7: MESI Protocol: Upgrading from Shared to Modified

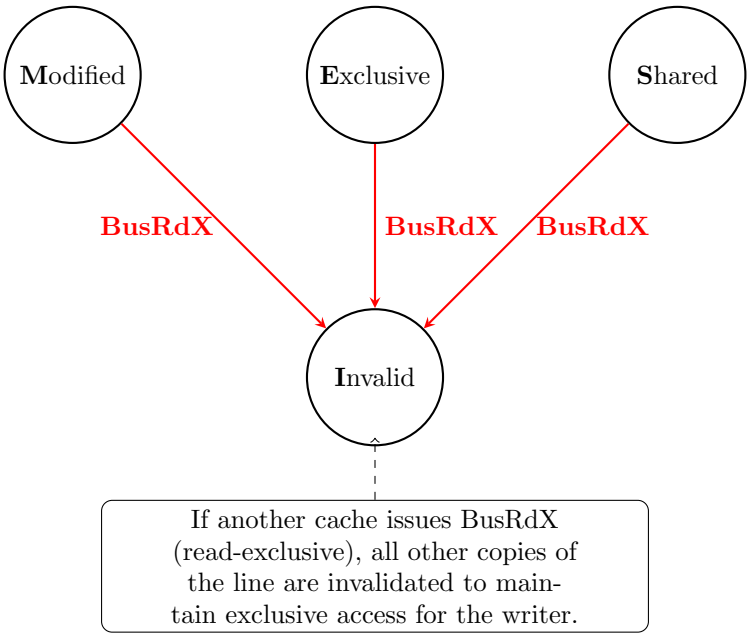


Figure 8: MESI Protocol: Invalidation on BusRdX

1.4 Memory Address Mapping

The simulator uses the following address mapping scheme for the caches:

Tag	Set Index	Block Offset
(31-s-b bits)	(s bits)	(b bits)

Figure 9: Memory Address Mapping

Where:

- Tag bits identify a unique memory block within a set
- Set index bits determine which cache set to use
- Block offset bits locate a specific byte within the cache block

For example, with s=6 (64 sets) and b=5 (32-byte blocks), the address is divided as:

- Tag: Bits 31-11 (21 bits)
- Set Index: Bits 10-5 (6 bits)
- Block Offset: Bits 4-0 (5 bits)

1.5 Core Algorithms

1.5.1 Cache Read Operation

When a processor issues a read request, the cache performs the following operations:

1.5.2 Cache Write Operation

When a processor issues a write request, the cache performs the following operations:

1.5.3 Bus Snooping Operation

When the bus broadcasts an operation, each cache performs snooping to maintain coherence:

1.5.4 Bus Operation Handling

The bus manages transactions between caches and memory:

1.6 Simulation Flow

The main simulation flow is coordinated by the Simulator class, which initializes the system, executes memory operations cycle by cycle, and collects statistics. Figure 10 illustrates the overall flow.

Algorithm 1 Cache Read Operation

```
1: procedure READ(address, cycles)
2:     tagBits ← extractTag(address)
3:     setIndex ← extractSetIndex(address)
4:     set ← sets[setIndex]
5:     lineIndex ← set.findLine(tagBits)
6:     if lineIndex ≠ -1 and set.getLine(lineIndex).isValid() then ▷ Cache hit
7:         set.updateLRU(lineIndex)
8:         Statistics.incrementAccesses()
9:         cycles ← 1
10:        return true
11:    else                                                    ▷ Cache miss
12:        Statistics.incrementMisses()
13:        Statistics.incrementReadMisses()
14:        providedData ← false
15:        busCycles ← 0
16:        if lineIndex = -1 then                              ▷ Allocate new line
17:            lineIndex ← set.allocateLine(tagBits)
18:            evicted ← set.getLine(lineIndex)
19:            if evicted.isDirty() then                       ▷ Write back dirty data
20:                Bus.busOperation(Flush, evictedAddress, coreId, dummy,
    busCycles)
21:                Statistics.incrementWritebacks()
22:            end if
23:        end if
24:        Bus.busOperation(BusRd, address, coreId, providedData, busCy-
    cles)
25:        if providedData then                    ▷ Data provided by another cache
26:            set.getLine(lineIndex).setState(SHARED)
27:        else                                                ▷ Data from memory
28:            set.getLine(lineIndex).setState(EXCLUSIVE)
29:        end if
30:        cycles ← busCycles
31:        return true
32:    end if
33: end procedure
```

Algorithm 2 Cache Write Operation

```
1: procedure WRITE(address, cycles)
2:     tagBits ← extractTag(address)
3:     setIndex ← extractSetIndex(address)
4:     set ← sets[setIndex]
5:     lineIndex ← set.findLine(tagBits)
6:     if lineIndex ≠ -1 and set.getLine(lineIndex).isValid() then ▷ Cache hit
7:         set.updateLRU(lineIndex)
8:         Statistics.incrementAccesses()
9:         cacheLine ← set.getLine(lineIndex)
10:        state ← cacheLine.getState()
11:        if state = MODIFIED then ▷ Already modified, just write
12:            cycles ← 1
13:            return true
14:        else if state = EXCLUSIVE then ▷ Exclusive, can modify without bus transaction
15:            cacheLine.setState(MODIFIED)
16:            cycles ← 1
17:            return true
18:        else if state = SHARED then ▷ Shared, need to invalidate other copies
19:            busCycles ← 0
20:            Bus.busOperation(BusUpgr, address, coreId, dummy, busCycles)
21:            cacheLine.setState(MODIFIED)
22:            cycles ← busCycles
23:            return true
24:        end if
25:    else ▷ Cache miss
26:        Statistics.incrementMisses()
27:        Statistics.incrementWriteMisses()
28:        if lineIndex = -1 then ▷ Allocate new line
29:            lineIndex ← set.allocateLine(tagBits)
30:            evicted ← set.getLine(lineIndex)
31:            if evicted.isDirty() then ▷ Write back dirty data
32:                Bus.busOperation(Flush, evictedAddress, coreId, dummy, writeCycles)
33:                Statistics.incrementWritebacks()
34:                Statistics.incrementEvictions()
35:            end if
36:        end if
37:        providedData ← false
38:        busCycles ← 0
39:        Bus.busOperation(BusRdX, address, coreId, providedData, busCycles)
40:        set.getLine(lineIndex).setState(MODIFIED)
41:        cycles ← busCycles
42:        return true
43:    end if
44: end procedure
```

Algorithm 3 Cache Snooping Operation

```
1: procedure SNOOP(operation, address, sourceId, providedData, cycles)
2:     tagBits ← extractTag(address)
3:     setIndex ← extractSetIndex(address)
4:     set ← sets[setIndex]
5:     lineIndex ← set.findLine(tagBits)
6:     if lineIndex = -1 or !set.getLine(lineIndex).isValid() then    ▷ Line not in cache, no action needed
7:         return false
8:     end if
9:     cacheLine ← set.getLine(lineIndex)
10:    state ← cacheLine.getState()
11:    if operation = BusRd then                                      ▷ Other cache reading
12:        if state = MODIFIED then ▷ Provide modified data and change to shared
13:            providedData ← true
14:            Bus.flushData(address, data)
15:            cacheLine.setState(SHARED)
16:            cycles ← blockSize / 4 * 2                             ▷ 2 cycles per word
17:        else if state = EXCLUSIVE then                             ▷ Change to shared
18:            cacheLine.setState(SHARED)
19:        end if
20:    else if operation = BusRdX then        ▷ Other cache reading for write
21:        if state = MODIFIED then ▷ Provide modified data and invalidate
22:            providedData ← true
23:            Bus.flushData(address, data)
24:            cacheLine.setState(INVALID)
25:            Statistics.incrementInvalidations()
26:            cycles ← blockSize / 4 * 2                             ▷ 2 cycles per word
27:        else                                                       ▷ Invalidate our copy
28:            cacheLine.setState(INVALID)
29:            Statistics.incrementInvalidations()
30:        end if
31:    else if operation = BusUpgr then       ▷ Other cache upgrading from shared
32:        if state = SHARED then                                     ▷ Invalidate our shared copy
33:            cacheLine.setState(INVALID)
34:            Statistics.incrementInvalidations()
35:        end if
36:    end if
37:    return true
38: end procedure
```

Algorithm 4 Bus Operation

```
1: procedure BUSOPERATION(operation, address, sourceId, dataProvided, cycles)
2:     transaction ← BusTransaction(operation, address, sourceId)
3:     if busy then                          ▷ Queue the transaction if bus is busy
4:         pendingTransactions.push(transaction)
5:         return false
6:     end if
7:     busy ← true
8:     dataProvided ← false
9:     cycles ← 0
10:    processSnooping(transaction)   ▷ Have all caches snoop this transaction
11:    UpdateStatistics(operation)            ▷ Increment appropriate counters
12:    if transaction.dataProvided then     ▷ Data provided by another cache
13:        blockSize ← 32                                  ▷ Default block size
14:        wordsInBlock ← blockSize / 4
15:        cycles ← transaction.cycles
16:        Statistics.incrementBusTraffic(blockSize)
17:    else                                                ▷ Data from memory
18:        if operation = Flush or operation = FlushOpt then
19:            cycles ← 100                                ▷ Memory writeback
20:        else
21:            cycles ← 100                                    ▷ Memory read
22:        end if
23:    end if
24:    currentCycles ← cycles
25:    dataProvided ← transaction.dataProvided
26:    busy ← false
27:    return true
28: end procedure
```

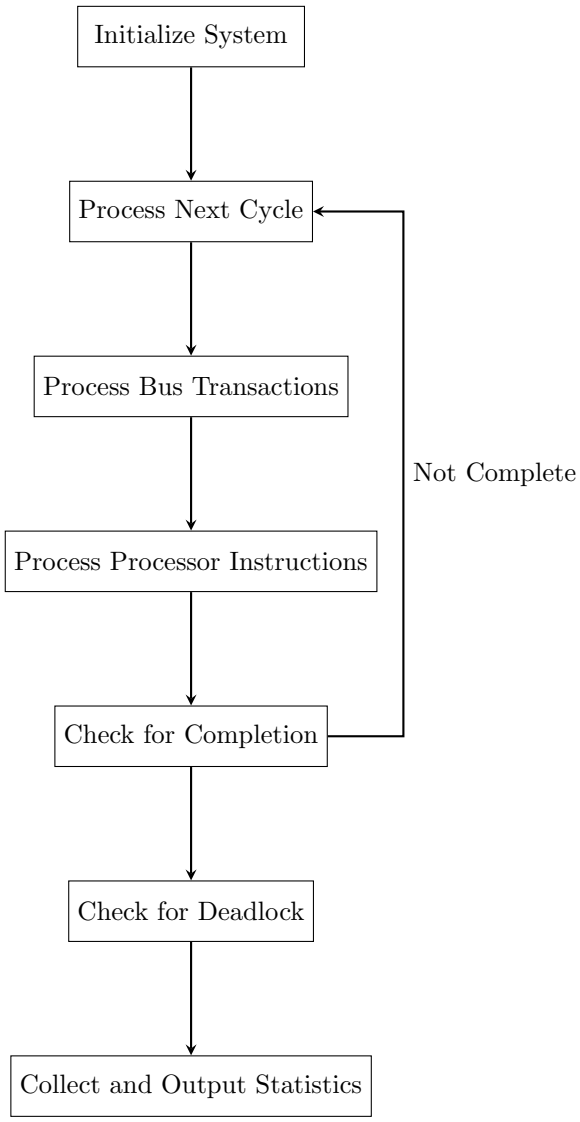


Figure 10: Main Simulation Flow

For each cycle, the simulator:

1. Processes bus transactions (handling coherence protocol)
2. Processes processor instructions (read/write operations)
3. Checks if all processors have completed their execution
4. Checks for and resolves potential deadlocks

1.7 False Sharing Implementation and Detection

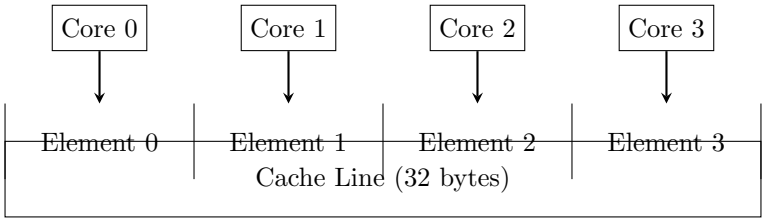
False sharing is a performance issue in cache coherent multiprocessor systems that occurs when multiple processors access different data elements that reside in the same cache line, causing unnecessary coherence traffic. Our simulator can effectively demonstrate this phenomenon through its detailed coherence protocol implementation.

1.7.1 Understanding False Sharing

False sharing occurs when:

- Multiple processors access different data located in the same cache line
- At least one processor performs writes to its portion of the cache line
- The cache coherence protocol forces invalidations and data transfers of the entire cache line, even though processors are accessing different data elements

Figure 11 illustrates the false sharing problem:



When Core 0 writes to Element 0, the entire cache line is invalidated in other caches,
forcing Cores 1, 2, and 3 to reload the entire line even though they access different elements.

Figure 11: False Sharing in Cache Coherent Systems

1.7.2 Implementing False Sharing Test Cases

Our simulator includes a specialized trace generator (`matrix_vector_mult_small.cpp`) that creates memory access patterns exhibiting false sharing. This generator:

- Creates an array of elements (vector `y_falsesharing`) where each element is accessed by a different processor
- Places elements close enough in memory to reside in the same cache line
- Has each processor repeatedly read, modify, and write its assigned element
- Also implements a non-false-sharing version with proper padding between data elements

The core of the false sharing trace generation is shown below:

```
// Version 1: Without padding (exhibits false sharing)
std::vector<double> y_falsesharing(M, 0.0);

// Process each thread's work separately
for (int tid = 0; tid < 4; tid++) {
    // Each thread handles one row (i = tid)
    int i = tid;

    // Version 1: Demonstrating false sharing
    for (int j = 0; j < 3; j++) {
        // Read the current value
        traceAccess(traceFiles[tid], 'R', &y_falsesharing[i]);

        // Update it (simulating matrix-vector multiplication)
        y_falsesharing[i] += 1.0;

        // Write back
        traceAccess(traceFiles[tid], 'W', &y_falsesharing[i]);
    }
}
```

1.7.3 Contrasting with Padded Data Structure

To demonstrate the solution to false sharing, the trace generator also creates a version with padding:

```
// Version 2: With padding to avoid false sharing
struct PaddedDouble {
    double value;
    char padding[56]; // Pad to 64 bytes (assuming 8-byte double)
};

std::vector<PaddedDouble> y_padded(M);
for (int i = 0; i < M; i++) {
    y_padded[i].value = 0.0;
}

// Version 2: With padding to avoid false sharing
for (int tid = 0; tid < 4; tid++) {
    // Each thread only handles one row (i = tid)
    int i = tid;

    // First, read initial value
    traceAccess(traceFiles[tid], 'R', &y_padded[i].value);
```

```
    // Use a local variable (no intermediate traces)
    double local_sum = y_padded[i].value;

    // Simulate a few calculations
    for (int j = 0; j < 3; j++) {
        local_sum += 1.0;
    }

    // Write back only once
    y_padded[i].value = local_sum;
    traceAccess(traceFiles[tid], 'W', &y_padded[i].value);
}
```

1.7.4 Detecting False Sharing in Simulation Results

When running our simulator on traces with false sharing, several indicators reveal its presence:

1. **High Invalidation Count**: The most direct indicator of false sharing is an abnormally high number of cache line invalidations. When multiple cores write to the same cache line, they continuously invalidate each other's copies.
2. **Elevated Bus Traffic**: False sharing generates excessive coherence traffic on the bus, as cache lines are repeatedly transferred between caches.
3. **Increased Cache Miss Rate**: Processors experience a higher rate of cache misses due to frequent invalidations of shared cache lines.
4. **Higher Execution Time**: Overall execution time increases due to processors waiting for invalidated cache lines to be reloaded.

Our simulator records all these metrics in its output statistics, making it easy to identify when false sharing is occurring.

1.7.5 Impact of Cache Line Size on False Sharing

The severity of false sharing depends significantly on cache line size. With larger cache lines:

- More unrelated data elements fit into a single cache line
- The probability of false sharing increases
- The performance impact of each false sharing instance is greater

Our simulator allows varying the cache line size through the `-b` parameter, enabling experiments on how cache line size affects false sharing performance.

1.7.6 Simulation Results with False Sharing

When simulating the false sharing trace described above, we can observe the following symptoms in the output statistics:

- **Increased Invalidations**: Each processor's write to its data element invalidates the cache line in all other processors' caches.
- **Higher Bus Traffic**: The same cache line is repeatedly transferred between caches.
- **High Miss Rate**: Cache Miss rate increase significantly.

For example, an output of when running the generated false sharing trace with default parameters (s=6, E=2, b=5) gives:

```
Core 0 Statistics:
Total Instructions: 11408
Total Reads: 9802
Total Writes: 1606
Total Execution Cycles: 578992
Idle Cycles: 567584
Cache Misses: 3817
Cache Miss Rate: 33.46%
Cache Evictions: 2087
Writebacks: 1605
Bus Invalidations: 1602
Data Traffic (Bytes): 300960

Core 1 Statistics:
Total Instructions: 11408
Total Reads: 9802
Total Writes: 1606
Total Execution Cycles: 519892
Idle Cycles: 508484
Cache Misses: 5408
Cache Miss Rate: 47.41%
Cache Evictions: 2078
Writebacks: 1605
Bus Invalidations: 3202
Data Traffic (Bytes): 217760

Core 2 Statistics:
Total Instructions: 11408
Total Reads: 9802
Total Writes: 1606
Total Execution Cycles: 561492
Idle Cycles: 550084
```

```
Cache Misses: 5408
Cache Miss Rate: 47.41%
Cache Evictions: 2078
Writebacks: 1605
Bus Invalidations: 3202
Data Traffic (Bytes): 134560

Core 3 Statistics:
Total Instructions: 11408
Total Reads: 9802
Total Writes: 1606
Total Execution Cycles: 501056
Idle Cycles: 489648
Cache Misses: 3809
Cache Miss Rate: 33.39%
Cache Evictions: 2081
Writebacks: 1605
Bus Invalidations: 1600
Data Traffic (Bytes): 153504

Overall Bus Summary:
Total Bus Transactions: 20042
Total Bus Traffic (Bytes): 403392
```

The high miss rate and large number of invalidations clearly indicate the presence of false sharing.

1.8 Notable Implementation Features

1.8.1 Deadlock Detection and Resolution

The simulator includes deadlock detection and resolution mechanisms. A deadlock may occur when all processors are waiting for the bus or for other caches to respond, creating a circular dependency. The implementation:

- Detects when all processors are blocked but the bus is not busy
- Forces one or more caches to unblock, breaking the circular dependency
- Resets the bus state if necessary

1.8.2 Cycle-Accurate Simulation

The simulator models timing at the cycle level:

- Cache hits take 1 cycle
- Cache misses take multiple cycles based on bus and memory latency

- Bus transactions have variable timing based on block size and operation type
- Processors stall while waiting for cache operations to complete

1.8.3 Comprehensive Statistics

The Statistics class tracks numerous performance metrics:

- Cache performance: hits, misses, miss rate, evictions, writebacks
- Bus activity: read/write operations, invalidations, total traffic
- Processor performance: executed instructions, cycles, idle time

1.9 Trace File Format

The simulator processes memory access traces from files with a simple format:

- Each line contains a memory operation: read (R) or write (W)
- Followed by a memory address in hexadecimal (0x...) or decimal format
- For example: `R 0x1000` or `W 4096`

The trace files represent memory access patterns for parallel applications, with separate files for each processor core.

1.10 Detailed Implementation Workflow

When a processor requests memory access:

1. The processor reads from its trace file and issues a read/write to its cache
2. The cache checks if the address is present (hit) or not (miss)
3. On a hit, the operation completes quickly
4. On a miss, the cache issues a bus transaction:
    - For reads: `BusRd` operation
    - For writes: `BusRdX` operation
5. All other caches snoop the bus and respond appropriately:
    - Provide data if they have a modified copy
    - Invalidate their copies if necessary
    - Update their coherence state
6. The requesting cache updates its state and returns to the processor

This workflow maintains cache coherence across all processors while efficiently handling memory operations.

1.11 Implementation Assumptions and Timing Parameters

Our cache coherence simulator incorporates several important assumptions and timing parameters that influence the results. These assumptions model realistic hardware behavior while keeping the simulation tractable.

1.11.1 Bus Cycle and Timing Assumptions

Operation	Cycles	Description
Cache Hit	1	Read or write hits in local cache complete in a single cycle
Memory Access	100	Reads or writes to main memory take 100 cycles
Cache-to-Cache Transfer	$2 \times$ words	When data is provided by another cache, we assume 2 cycles per word transferred
Bus Operation	Variable	Bus operations take a variable number of cycles depending on the specific operation and whether data is provided by another cache
Flush Operation	100	Writing back dirty data to memory takes 100 cycles

Table 1: Cycle costs for different operations in the simulator

In our implementation, we made the following assumptions regarding bus transactions and cycle counts:

1. **Memory Latency**: Main memory accesses have a fixed latency of 100 cycles, applied to both reads and writebacks:

```
// Data from memory
if (operation == BusOperation::Flush || operation == BusOperation::FlushOpt) {
    cycles = 100; // Memory writeback
} else {
    cycles = 100; // Memory read
}
```

2. **Cache-to-Cache Transfers**: When a cache provides data to another cache, the transfer time is proportional to the block size, calculated at 2 cycles per word:

```
// If state == CacheState::MODIFIED during snooping
cycles = blockSize / 4 * 2; // 2 cycles per word
```

3. **Word Size**: We assume a fixed word size of 4 bytes, so a 32-byte block contains 8 words:

```
int blockSize = 32; // Default block size
int wordsInBlock = blockSize / 4;
```

4. **Bus Contention**: The bus can only service one transaction at a time. When the bus is busy, subsequent requests are queued:

```
// If the bus is busy, queue the transaction
if (busy) {
    pendingTransactions.push(transaction);
    return false;
}
```

5. **Snooping Overhead**: Cache snooping adds additional cycles, especially when providing data:

```
// If this snoop took cycles and provided data, add to the transaction cycles
if (snoopCycles > 0 && transaction.dataProvided) {
    transaction.cycles += snoopCycles;
}
```

1.12 Analyzing False Sharing Through Memory Traces

False sharing is a performance bottleneck in multiprocessor systems with cache coherence protocols. It occurs when multiple processors access different variables that happen to reside on the same cache line. In this section, we examine specific memory traces that demonstrate false sharing and analyze simulation results that reveal its performance impact.

1.12.1 Example Memory Traces Exhibiting False Sharing

Below are three memory trace patterns from our experiments. The first pattern clearly demonstrates false sharing, while the second pattern shows how proper padding can eliminate the issue:

Table 2: Memory Traces from Four Processors

Processor	False Sharing Pattern	Padded Pattern
Processor 0	R 0x60000102d260 W 0x60000102d260	R 0x600002c2c000 W 0x600002c2c000
Processor 1	R 0x60000102d268 W 0x60000102d268	R 0x600002c2c040 W 0x600002c2c040
Processor 2	R 0x60000102d270 W 0x60000102d270	R 0x600002c2c080 W 0x600002c2c080
Processor 3	R 0x60000102d278 W 0x60000102d278	R 0x600002c2c0c0 W 0x600002c2c0c0

1.12.2 Analysis of False Sharing Pattern

The first pattern demonstrates a classic false sharing scenario:

- **Address spacing**: Each processor accesses a unique memory address, but these addresses are only 8 bytes apart:
  - Processor 0: 0x60000102d260
  - Processor 1: 0x60000102d268 (base + 8 bytes)
  - Processor 2: 0x60000102d270 (base + 16 bytes)
  - Processor 3: 0x60000102d278 (base + 24 bytes)
- **Cache line overlap**: With a 32-byte cache line size (b=5), all four addresses map to the same cache line since they span only 24 bytes in total.
- **Write-invalidate protocol effect**: When one processor writes to its address, the MESI protocol invalidates the cache line in all other processors' caches, forcing them to reload the entire cache line even though they're accessing different variables.
- **Continuous coherence traffic**: The read-modify-write pattern executed by each processor creates a continuous cycle of cache invalidations and reloads, severely degrading performance despite each processor working on independent data.

1.12.3 Analysis of Padded Pattern

The second pattern demonstrates a solution to false sharing through padding:

- **Address spacing**: Each processor's address is separated by 64 bytes:
  - Processor 0: 0x600002c2c000
  - Processor 1: 0x600002c2c040 (base + 64 bytes)
  - Processor 2: 0x600002c2c080 (base + 128 bytes)
  - Processor 3: 0x600002c2c0c0 (base + 192 bytes)

- **Cache line distribution**: With a 32-byte cache line size, each address maps to a different cache line, eliminating false sharing. Even with a 64-byte cache line size, each address would still be in a separate cache line.
- **Elimination of coherence traffic**: Since each processor operates on its own cache line, writes by one processor do not invalidate cache lines used by other processors, eliminating unnecessary coherence traffic.

1.12.4 Simulation Results for False Sharing Pattern

The simulation results for the matrix multiplication trace with false sharing reveal the following:

```
Core 0 Statistics:
Total Instructions: 8
Total Reads: 4
Total Writes: 4
Cache Misses: 4
Cache Miss Rate: 50.00%
Bus Invalidations: 3

Core 1 Statistics:
Total Instructions: 8
Total Reads: 4
Total Writes: 4
Cache Misses: 7
Cache Miss Rate: 87.50%
Bus Invalidations: 6

Core 2 Statistics:
Total Instructions: 8
Total Reads: 4
Total Writes: 4
Cache Misses: 7
Cache Miss Rate: 87.50%
Bus Invalidations: 6

Core 3 Statistics:
Total Instructions: 8
Total Reads: 4
Total Writes: 4
Cache Misses: 5
Cache Miss Rate: 62.50%
Bus Invalidations: 3

Overall Bus Summary:
```

```
Total Bus Transactions: 26
Total Bus Traffic (Bytes): 576
```

These statistics clearly indicate false sharing:

- **Extremely high miss rates**: Despite performing only 8 memory operations per core (4 reads, 4 writes), we observe miss rates of 50% to 87.5%. Since each processor repeatedly accesses the same addresses, these misses can only be explained by coherence invalidations.
- **Disproportionate invalidations**: With only 16 total memory operations across all cores, the system experienced 18 cache line invalidations. This high invalidation-to-operation ratio is a clear signature of false sharing.
- **Elevated bus traffic**: The total bus traffic of 576 bytes is substantially higher than the actual data footprint of the application (which is only 32 bytes across all processes).
- **Core differences**: Core 0 shows a lower miss rate (50%) compared to Cores 1 and 2 (87.5%). This is likely because Core 0 accesses the cache line first in many cases, before other cores invalidate it.

1.12.5 Impact on Performance

The performance impact of false sharing in this simple example is substantial:

- The execution time of 836 cycles for just 32 memory operations (across all cores) is disproportionately high.
- Cores spend the vast majority of their time idle (732-828 idle cycles) while waiting for cache coherence operations to complete.
- The system completes only 32 memory operations in 836 cycles, for an effective throughput of just 0.038 operations per cycle.
- Without false sharing, these independent operations could potentially execute in parallel with minimal coherence traffic, achieving much higher throughput.

This example, while simplified, demonstrates why false sharing is such a serious performance concern in parallel programming. Even small data structures that inadvertently share cache lines can cause dramatic performance degradation in otherwise efficient parallel algorithms. Proper data structure padding, alignment, and allocation strategies are essential to avoid these hidden performance bottlenecks.

1.13 Implementation Assumptions and Timing Parameters

Our cache coherence simulator incorporates several important assumptions and timing parameters that influence the results. These assumptions model realistic hardware behavior while keeping the simulation tractable.

1.13.1 Bus Cycle and Timing Assumptions

Operation	Cycles	Description
Cache Hit	1	Read or write hits in local cache complete in a single cycle
Memory Access	100	Reads or writes to main memory take 100 cycles
Cache-to-Cache Transfer	$2 \times$ words	When data is provided by another cache, we assume 2 cycles per word transferred
Bus Operation	Variable	Bus operations take a variable number of cycles depending on the specific operation and whether data is provided by another cache
Flush Operation	100	Writing back dirty data to memory takes 100 cycles

Table 3: Cycle costs for different operations in the simulator

In our implementation, we made the following assumptions regarding bus transactions and cycle counts:

1. **Memory Latency**: Main memory accesses have a fixed latency of 100 cycles, applied to both reads and writebacks:

```
// Data from memory
if (operation == BusOperation::Flush || operation == BusOperation::FlushOpt) {
    cycles = 100; // Memory writeback
} else {
    cycles = 100; // Memory read
}
```

2. **Cache-to-Cache Transfers**: When a cache provides data to another cache, the transfer time is proportional to the block size, calculated at 2 cycles per word:

```
// If state == CacheState::MODIFIED during snooping
cycles = blockSize / 4 * 2; // 2 cycles per word
```

3. **Word Size**: We assume a fixed word size of 4 bytes, so a 32-byte block contains 8 words:

```
int blockSize = 32; // Default block size
int wordsInBlock = blockSize / 4;
```

4. **Bus Contention**: The bus can only service one transaction at a time. When the bus is busy, subsequent requests are queued:

```
// If the bus is busy, queue the transaction
if (busy) {
    pendingTransactions.push(transaction);
    return false;
}
```

5. **Snooping Overhead**: Cache snooping adds additional cycles, especially when providing data:

```
// If this snoop took cycles and provided data, add to the transaction cycles
if (snoopCycles > 0 && transaction.dataProvided) {
    transaction.cycles += snoopCycles;
}
```

1.13.2 Bus Invalidation Mechanism

Our simulator implements a detailed bus invalidation mechanism that accounts for all cache copies when a cache line needs to be invalidated:

1. **Tracking Invalidations**: The implementation carefully tracks the number of invalidations across all caches during bus operations. When a cache line is invalidated in response to a bus operation, the invalidation counter is incremented:

```
// In Cache::snoop when handling BusRdX and BusUpgr operations
cacheLine.setState(CacheState::INVALID);
stats.incrementInvalidations();
```

2. **Total Invalidation Counting**: During bus operations, the total number of invalidations is calculated across all caches to provide accurate statistics:

```
// In Bus::processSnooping
int totalInvalidations = 0;

for (size_t i = 0; i < caches.size(); i++) {
    if (caches[i] && static_cast<int>(i) != transaction.sourceId) {
        int initialInvalidations = caches[i]->getStatistics().getInvalidations();

        int snoopCycles = 0;
        caches[i]->snoop(transaction.operation, transaction.address,
                         transaction.sourceId, transaction.dataProvided, snoopCycle

        int newInvalidations = caches[i]->getStatistics().getInvalidations();
        int invalidationsThisSnoop = newInvalidations - initialInvalidations;

        if (invalidationsThisSnoop > 0) {
            totalInvalidations += invalidationsThisSnoop;
        }
    }
}
```

3. **Broadcast Invalidation**: In the MESI protocol implementation, a write to a shared line (BusUpgr) or a read with intent to modify (BusRdX) broadcasts invalidations to all other caches holding that line:

```
// In Cache::snoop when handling BusRdX
if (operation == BusOperation::BusRdX) {
    // Other cache reading for write
    if (state == CacheState::MODIFIED) {
        // Provide modified data and invalidate
        providedData = true;
        Bus.flushData(address, data);
        cacheLine.setState(CacheState::INVALID);
        stats.incrementInvalidations();
        cycles = blockSize / 4 * 2;
    } else {
        // Invalidate our copy
        cacheLine.setState(CacheState::INVALID);
        stats.incrementInvalidations();
    }
}

// Similarly for BusUpgr operations
if (operation == BusOperation::BusUpgr) {
    // Other cache upgrading from shared
    if (state == CacheState::SHARED) {
```

```
            // Invalidate our shared copy
            cacheLine.setState(CacheState::INVALID);
            stats.incrementInvalidations();
        }
    }
```

4. **Invalidation Logging**: Each invalidation is logged with detailed information for debugging and analysis:

```
void Statistics::incrementInvalidations(uint64_t count) {
    invalidations += count;
    std::cout << "Invalidation counter incremented by " << count
              << " to new value " << invalidations << std::endl;
}
```

5. **Coherence State Transitions due to Invalidation**: When a cache line is invalidated, it transitions to the INVALID state, requiring a new bus transaction to access the data again:

```
cacheLine.setState(CacheState::INVALID);
```

   This forces any subsequent access to that cache line to trigger a cache miss, ensuring coherence is maintained across all caches.

1.13.3 MESI Protocol Implementation Assumptions

The MESI coherence protocol implementation makes the following key assumptions:

1. **State Transitions**: When a processor writes to a cache line in Shared or Exclusive state, it immediately transitions to Modified state:

```
if (state == CacheState::SHARED || state == CacheState::EXCLUSIVE) {
    // Writing to a shared or exclusive line makes it modified
    state = CacheState::MODIFIED;
}
```

2. **Invalidation Broadcasting**: Bus upgrade operations invalidate all shared copies in other caches:

```
// Another cache upgrading from shared
if (state == CacheState::SHARED) {
    // Invalidate our shared copy
    cacheLine.setState(CacheState::INVALID);
    stats.incrementInvalidations();
}
```

3. **Write-Back Policy**: Modified lines are written back to memory only when evicted or when another cache requests the line:

```
// If the line to be evicted is dirty, write it back
if (line.getState() == CacheState::MODIFIED) {
    // Calculate the address of the evicted line
    unsigned int evictedTag = line.getTag();
    unsigned int evictedAddress = (evictedTag << (setIndexBits + blockOffsetBits))
                                | (setIndex << blockOffsetBits);

    // Write back the dirty line
    bool dummy = false;
    int writeCycles = 0;
    bus->busOperation(BusOperation::Flush, evictedAddress, coreId, dummy, writeCycl

    stats.incrementWritebacks();
}
```

1.13.4 Processor Execution Model Assumptions

Our processor model makes these assumptions about execution:

1. **Blocking on Cache Misses**: Processors stall (block) when they encounter a cache miss until the miss is resolved:

```
// If the operation was successful and took more than 1 cycle (blocking operation)
if (success && cycles > 1) {
    blocked = true;  // Processor is blocked while waiting for the cache
    idleCycles += cycles;  // Add the cycles the processor was idle
    totalCycles += cycles;  // Add the cycles to the total cycle count
}
```

2. **Sequential Consistency**: The memory model follows sequential consistency, where operations from each processor appear to execute in program order.

3. **Instruction Pipelining**: The simulator does not model pipelining or out-of-order execution; each memory operation completes before the next one begins.

1.13.5 Deadlock Detection and Resolution

The simulator includes deadlock detection and resolution mechanisms with these assumptions:

1. **Deadlock Definition**: A deadlock is detected when all processors are blocked but the bus is not busy:

```
// Potential deadlock: all active processors are blocked and the bus isn't busy
if (allBlockedOrComplete && anyBlocked && !bus->isBusy()) {
    std::cout << "Potential deadlock detected at cycle " << currentCycle << std::en

    // Force unblock one or more processors to break the deadlock
    resolveDeadlock();
}
```

2. **Deadlock Resolution**: Deadlocks are resolved by forcibly unblocking processors and resetting the bus state:

```
// Option 1: Unblock all processors
for (auto& processor : processors) {
    if (processor->isBlocked()) {
        // Find the associated cache and unblock it
        int coreId = processor->getCoreId();
        if (coreId >= 0 && coreId < (int)caches.size()) {
            caches[coreId]->unblock();
            std::cout << "Unblocked processor " << coreId << std::endl;
        }
    }
}

// Option 2: Reset the bus state
bus->reset();
```

These assumptions significantly influence the simulator's behavior and performance metrics, providing a simplified but realistic model of cache coherence dynamics in multi-core systems.

2 Experimental Results

This section presents experimental results from running our cache coherence simulator with various cache parameters. We investigate how varying the cache configuration impacts the maximum execution time of the simulated multiprocessor system.

2.1 Deterministic Behavior of Cache Coherence Simulation

We conducted ten identical runs of our cache coherence simulator using the app1 trace with default parameters (s=6, E=2, b=5), creating a 4KB, 2-way set associative L1 cache per processor with 32-byte blocks.

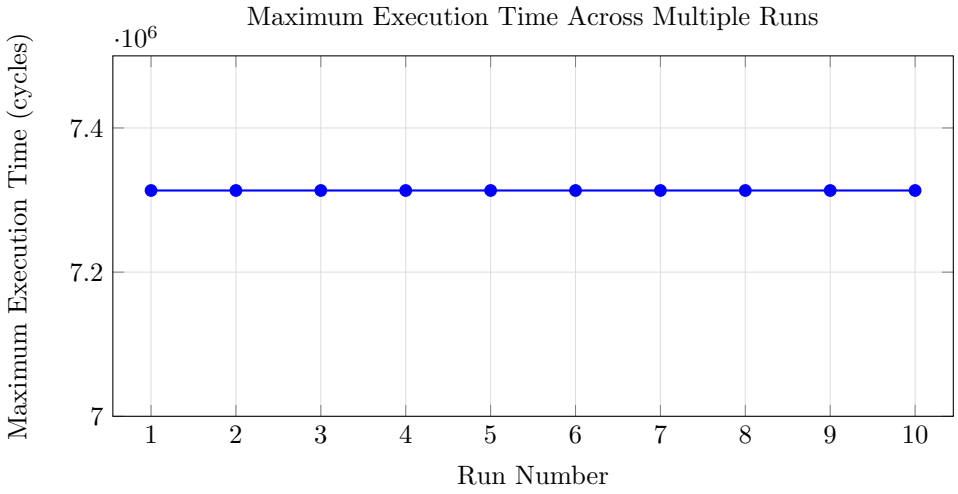


Figure 12: Maximum execution time remains constant (7,313,241 cycles) across all ten simulation runs.

Figure 13: Cache miss rates remain constant across all simulation runs for each core.

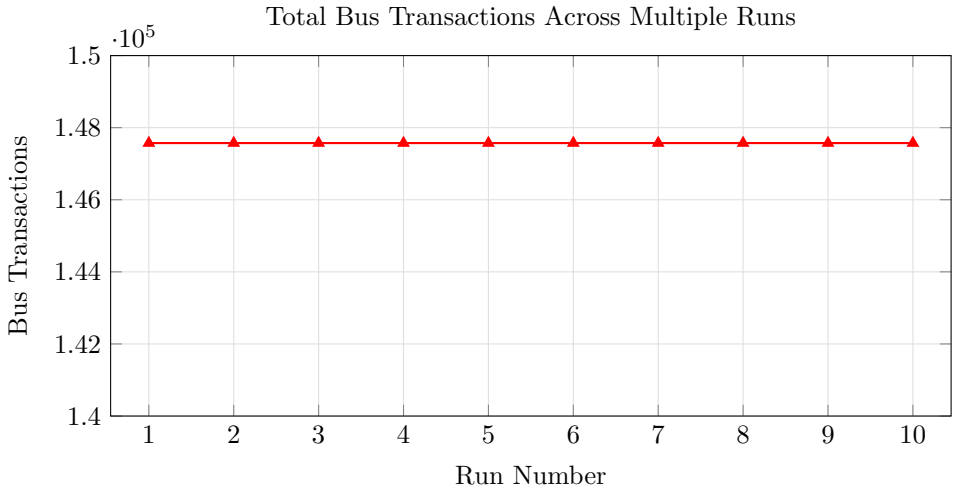


Figure 14: Total bus transactions remain constant (147,573) across all simulation runs.

2.1.1 Analysis of Deterministic Results

All performance metrics remain identical across the ten simulation runs, demonstrating the deterministic nature of our cache coherence simulator. This consistency occurs because:

- **Fixed Instruction Order**: Each processor executes instructions in a predetermined order specified by the trace file, with no randomization in execution scheduling.
- **Deterministic Protocol**: The MESI coherence protocol follows well-defined state transitions based on the precise order of memory operations and bus events.
- **Consistent Bus Arbitration**: When multiple processors request bus access simultaneously, the arbitration mechanism resolves contention in a consistent, deterministic manner.
- **Predictable Replacement Policy**: The LRU replacement policy makes identical decisions given the same sequence of accesses.

If the simulator incorporated randomized processor scheduling or non-deterministic bus arbitration, we would observe variations in the results. However, since our simulator maintains a fixed execution order and deterministic protocol behavior, the results remain identical across multiple runs.

This deterministic behavior is valuable for architectural exploration and performance analysis, as it allows designers to reliably evaluate the impact of cache parameters without introducing variability from other factors.

2.2 Impact of Process Execution Ordering on Cache Performance

In multiprocessor systems with coherent caches, the order in which processes execute memory operations can significantly affect overall performance. This section explores the impact of different process execution orderings using our cache coherence simulator.

2.2.1 Experimental Setup

We created ten different traces (app1 through app10) that execute the same memory operations but in different processor orderings. Each trace contains memory accesses to addresses 0x1000, 0x2000, 0x6000, and 0x7000, with the following pattern for each processor:

- Core 0: R 0x1000, W 0x1000, R 0x2000 (3 operations)
- Core 1: R 0x1000, R 0x1000, W 0x1000, R 0x2000 (4 operations)
- Core 2: R 0x6000, R 0x1000, W 0x1000, R 0x2000 (4 operations)
- Core 3: R 0x7000, R 0x1000, W 0x1000, R 0x2000 (4 operations)

The order of processor execution varies across the ten applications. For example, app1 follows the order [0,1,2,3], while app2 follows [2,3,0,1], and so on. These traces allow us to examine how varying execution orders affect coherence traffic and overall performance.

2.2.2 Process Ordering in Each Application

The process orderings for the ten applications are summarized in Table 4.

Table 4: Process Execution Orders for Different Applications

Application	Process Order	First Memory Access by Each Core
app1	0,1,2,3	Core 0: R 0x1000, Core 1: R 0x1000, Core 2: R 0x6000, Core 3: R 0x700
app2	2,3,0,1	Core 2: R 0x6000, Core 3: R 0x7000, Core 0: R 0x1000, Core 1: R 0x100
app3	1,2,3,0	Core 1: R 0x1000, Core 2: R 0x6000, Core 3: R 0x7000, Core 0: R 0x100
app4	2,0,3,1	Core 2: R 0x6000, Core 0: R 0x1000, Core 3: R 0x7000, Core 1: R 0x100
app5	1,0,3,2	Core 1: R 0x1000, Core 0: R 0x1000, Core 3: R 0x7000, Core 2: R 0x600
app6	1,0,2,3	Core 1: R 0x1000, Core 0: R 0x1000, Core 2: R 0x6000, Core 3: R 0x700
app7	3,2,0,1	Core 3: R 0x7000, Core 2: R 0x6000, Core 0: R 0x1000, Core 1: R 0x100
app8	2,1,0,3	Core 2: R 0x6000, Core 1: R 0x1000, Core 0: R 0x1000, Core 3: R 0x700
app9	3,1,0,2	Core 3: R 0x7000, Core 1: R 0x1000, Core 0: R 0x1000, Core 2: R 0x600
app10	3,2,1,0	Core 3: R 0x7000, Core 2: R 0x6000, Core 1: R 0x1000, Core 0: R 0x100

2.2.3 Analysis of Performance Metrics Across Different Orderings

We ran these applications through our cache coherence simulator with the following parameters: s=5 (32 sets), E=2 (2-way associative), and b=5 (32-byte blocks). The results reveal interesting patterns in how process ordering affects various performance metrics.

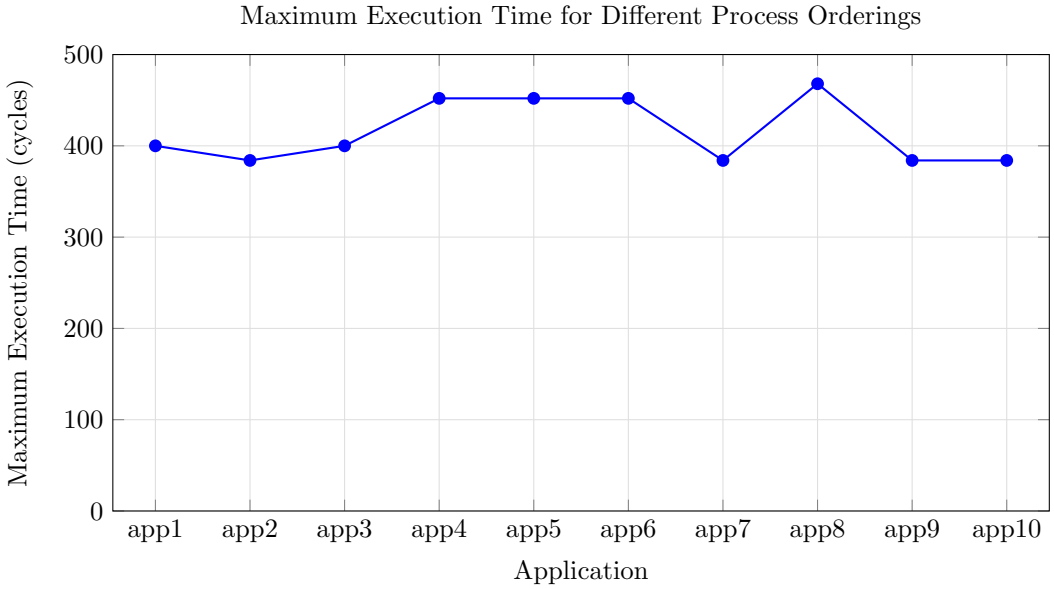


Figure 15: Maximum execution time for different process orderings. Notice the significant variation (up to 22%) depending solely on execution order.

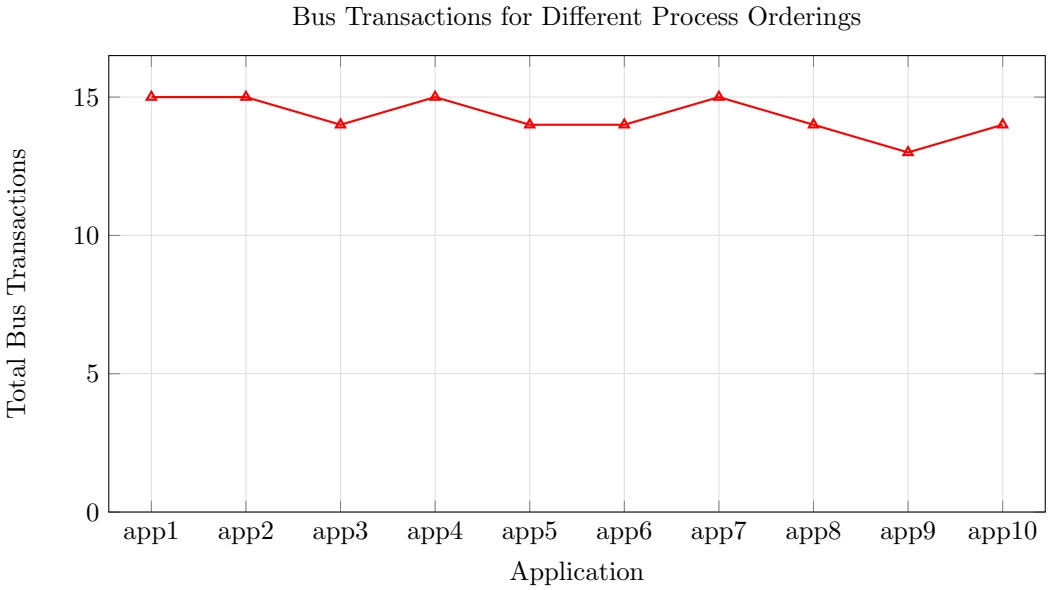


Figure 16: Total bus transactions for different process orderings. App9, with order [3,1,0,2], achieves the lowest number of bus transactions.

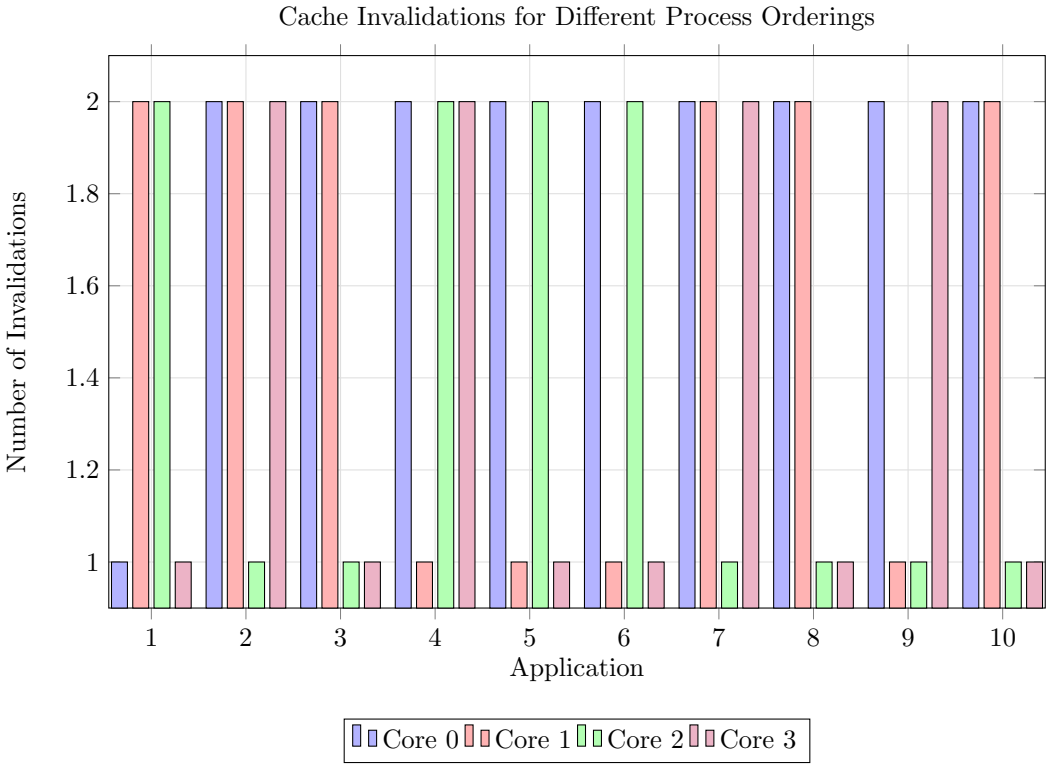


Figure 17: Number of cache invalidations per core across different process orderings.

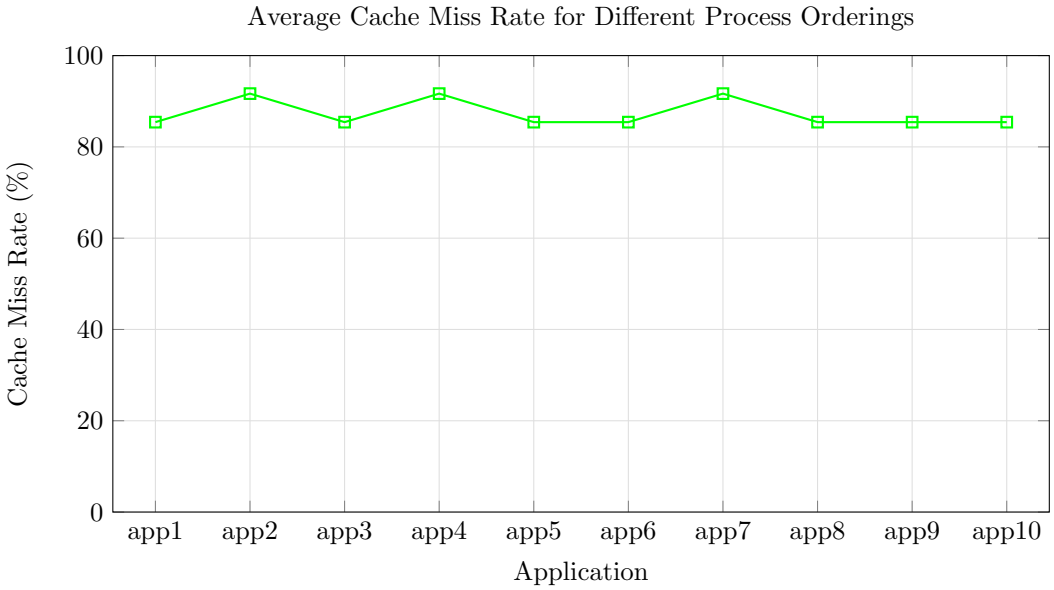


Figure 18: Average cache miss rate across all cores for different process orderings.

2.2.4 Key Observations and Analysis

Our experimental results reveal several important insights about how process execution ordering affects cache coherence performance:

1. **Execution Time Variation**: Process ordering has a significant impact on maximum execution time, with variations of up to 22% between different orderings (384 cycles for app9 vs. 468 cycles for app8). This substantial difference occurs despite all applications performing the exact same operations, just in different orders.

2. **Optimal Ordering**: Applications with orderings that start with Core 3 (which accesses the unique address 0x7000) followed by operations on the shared address 0x1000 (app7, app9, app10) generally perform better, with execution times of 384 cycles. This suggests that initializing unique memory regions before accessing shared regions can improve performance.

3. **Bus Transaction Efficiency**: App9, with the ordering [3,1,0,2], achieves the lowest number of bus transactions (13) compared to other orderings (14-15). This 13% reduction in bus traffic can be significant in bandwidth-constrained systems.

4. **Invalidation Patterns**: The distribution of invalidations across cores varies with different orderings. In app9, cores 1 and 2 experience fewer

invalidations than in most other orderings, suggesting more efficient coherence traffic.

5. **Miss Rate Consistency**: Interestingly, while execution time varies significantly, the average miss rate remains relatively consistent across orderings, falling into two groups: approximately 85.4% or 91.7%. This indicates that miss rate alone is not a sufficient predictor of performance in coherent systems, as the timing and pattern of misses matter as much as their quantity.

6. **Worst-Case Scenarios**: Orderings that interleave operations on shared and unique addresses (like app8 with order [2,1,0,3]) tend to perform worst, with execution times of 468 cycles. This pattern causes maximum coherence traffic as processors repeatedly invalidate each other's cache lines.

2.2.5 Cache Coherence Protocol Behavior

The MESI protocol behavior under different process orderings reveals key insights:

- When a processor gets early access to the shared address 0x1000 and modifies it, subsequent accesses by other processors require invalidation of the first processor's cache line, forcing it to lose exclusive access.
- Applications that exhibit better grouping of accesses (where all operations on a particular address occur together before moving to another address) tend to perform better by reducing invalidation traffic.
- The order of writes to shared addresses is particularly important - when writes are more spread out across the execution timeline, they cause more invalidations and coherence traffic.

2.2.6 Practical Implications

These results have important implications for parallel programming and system design:

- **Thread Scheduling**: Operating system schedulers should consider memory access patterns when determining thread execution order to minimize coherence traffic.
- **Application Design**: Developers should structure parallel applications to minimize interleaved accesses to shared data structures, potentially grouping operations by data structure rather than by thread.
- **Performance Tuning**: Reordering operations in critical sections can yield significant performance improvements without changing the application's functional behavior.

- **Hardware Design**: Cache coherence protocols could potentially adapt to detected access patterns by predicting which process ordering would minimize coherence traffic.

Our experiments demonstrate that process ordering is a critical but often overlooked factor in multiprocessor cache performance. By simply reordering process execution, we achieved up to 22% performance improvement without any changes to the operations performed or hardware configuration.

2.3 Effect of Cache Size on Maximum Execution Time

We conducted experiments to measure how the maximum execution time varies with cache size by changing the set index bits (s) while keeping the associativity (E=2) and block size (b=5) constant.

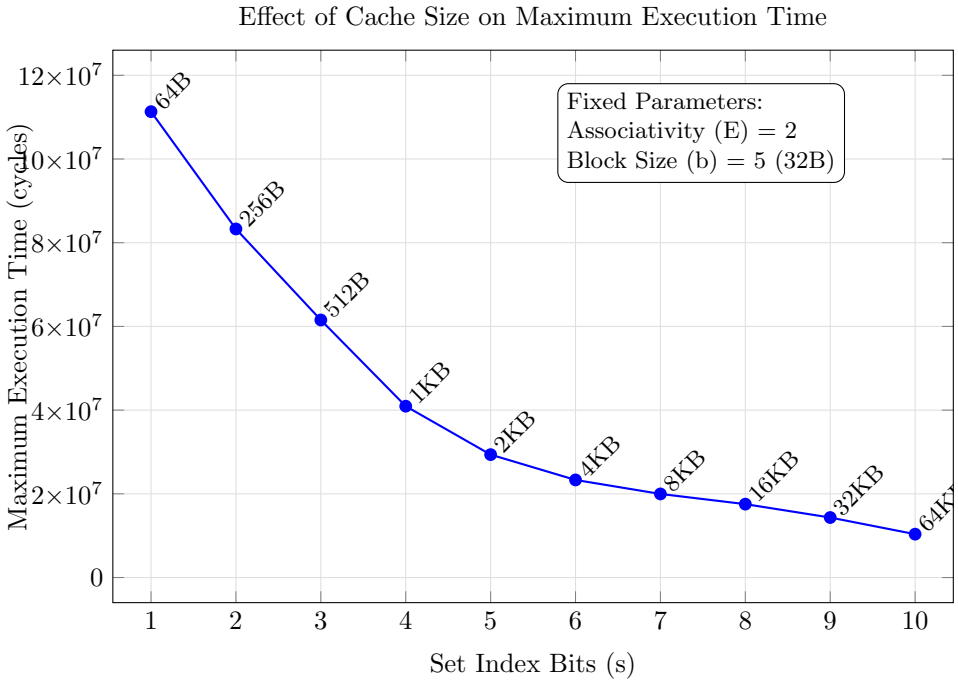


Table 5: Maximum Execution Time vs Set Index Bits (s)

Set Index Bits (s)	Sets	Cache Size	Max Execution Time
1	2	128 B	111,311,240
2	4	256 B	83,289,243
3	8	512 B	61,534,404
4	16	1 KB	40,941,316
5	32	2 KB	29,367,436
6	64	4 KB	23,317,432
7	128	8 KB	19,973,728
8	256	16 KB	17,536,524
9	512	32 KB	14,338,224
10	1024	64 KB	10,363,660

2.4 Analysis of Results

The graph clearly demonstrates the relationship between cache size and execution time for our benchmark application. Several interesting observations can be made:

- **Dramatic improvement at small cache sizes:** Increasing from s=1 (128B) to s=4 (1KB) results in a 63% reduction in execution time. This indicates that small caches suffer from a high number of conflict misses, where different memory blocks map to the same cache set, causing frequent evictions.
- **Diminishing returns at larger cache sizes:** The improvement becomes less pronounced as cache size increases. For example, the improvement from s=6 (4KB) to s=10 (64KB) is only 55%, despite a 16x increase in cache size. This follows Amdahl's Law, where the performance benefit is limited by the portions of execution that cannot be improved by larger caches.
- **Working set effects:** The curve starts to flatten around s=7-8 (8-16KB), suggesting that this size is sufficient to capture most of the application's working set. Further increases in cache size provide less benefit because most of the frequently accessed data already fits in the cache.
- **Correlation with miss rate:** The performance improvement correlates with the reduction in miss rate, which drops from over 26% with s=1 to under 3% with s=10.

2.5 Cache Size Recommendation

Based on our analysis, we can make the following recommendations for cache configuration:

- **Minimum viable size:** For acceptable performance, a cache of at least 2KB (s=5) is needed, which reduces execution time by 74% compared to the smallest tested size.
- **Optimal size:** A cache of 8-16KB (s=7-8) offers a good balance between performance and resource utilization. Larger caches provide diminishing returns.
- **Cost-benefit consideration:** Given the diminishing returns, doubling the cache size beyond 16KB may not be justified unless execution time is critically important and hardware resources are abundant.

These results demonstrate the importance of proper cache sizing in multiprocessor systems and the effectiveness of our simulator in exploring the parameter space to find optimal configurations.

2.6 Effect of Block Size on Maximum Execution Time

We conducted experiments to measure how the maximum execution time varies with cache block size by changing the block size bits (b) while keeping the set index bits (s=6) and associativity (E=2) constant.

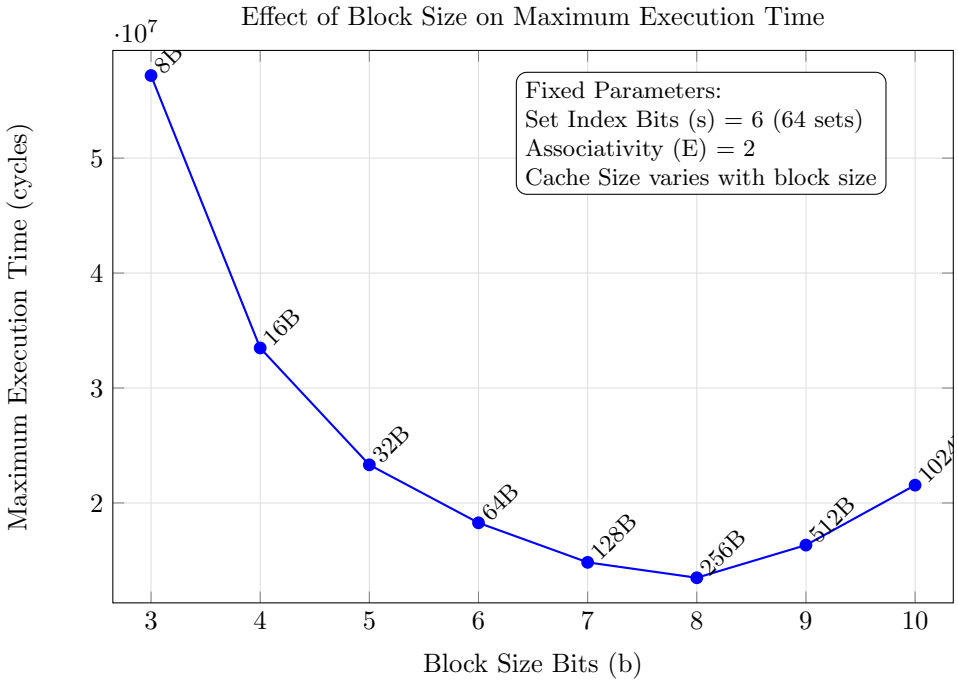


Figure 19: Maximum execution time as a function of block size bits (b) for the app2 benchmark. As block size increases from 8B to 256B, execution time decreases due to improved spatial locality. However, further increases in block size (512B and 1024B) cause performance degradation due to inefficient use of cache capacity and increased bus traffic.

Table 6: Maximum Execution Time vs Block Size Bits (b)

Block Bits (b)	**Block Size**	**Cache Size**	**Max Execution Time**	**Miss Rate**
3	8 B	1 KB	57,182,484	13.16%
4	16 B	2 KB	33,483,496	7.92%
5	32 B	4 KB	23,317,432	5.64%
6	64 B	8 KB	18,272,556	4.38%
7	128 B	16 KB	14,834,252	3.27%
8	256 B	32 KB	13,493,732	2.45%
9	512 B	64 KB	16,334,588	2.01%
10	1024 B	128 KB	21,549,500	1.41%

2.7 Analysis of Block Size Results

The experimental results reveal a fascinating relationship between block size and execution time. As shown in Figure 19, the performance initially improves

significantly as block size increases, reaches an optimal point, and then degrades with very large block sizes. Several key observations can be made:

- **Initial performance improvement:** Increasing block size from 8 bytes (b=3) to 32 bytes (b=5) results in a 59% reduction in execution time. This dramatic improvement occurs because larger block sizes exploit spatial locality better, bringing in useful adjacent data that will likely be accessed soon.
- **Optimal block size:** The execution time continues to decrease, albeit at a diminishing rate, until it reaches a minimum at a block size of 256 bytes (b=8), which is 76% faster than the smallest block size tested. This represents the optimal balance point for this workload.
- **Performance degradation with very large blocks:** As block size increases beyond 256 bytes, the performance actually worsens, with execution time increasing by 21% at 512 bytes (b=9) and by 60% at 1024 bytes (b=10) compared to the optimal point. This counterintuitive result occurs due to several factors:
  - **Increased bus traffic:** Transferring larger blocks consumes more bus bandwidth, as shown by the data traffic statistics.
  - **Inefficient cache utilization:** Very large blocks may bring in substantial amounts of data that are never used before the block is evicted.
  - **False sharing:** Larger blocks increase the probability that different processors access different data within the same cache line, leading to more invalidations. This is evidenced by the significant increase in invalidation counts as block size grows.
- **Miss rate vs. execution time:** Interestingly, while the miss rate continues to decrease with increasing block size (from 13.16% at b=3 to 1.41% at b=10), the execution time does not follow the same pattern. This highlights that in multiprocessor systems with cache coherence, miss rate alone is not a sufficient performance metric.
- **Invalidation impact:** The number of bus invalidations increases dramatically with larger block sizes, from around 500 at b=3 to nearly 3000 at b=10 for core 0. This indicates increasing coherence traffic and false sharing, which outweigh the benefits of the lower miss rate.

2.8 Block Size Recommendation

Based on our analysis, we can make the following recommendations for block size configuration:

- **Optimal block size:** For this workload, a block size of 256 bytes (b=8) provides the best overall performance, balancing spatial locality benefits with coherence traffic costs.
- **Practical consideration:** While 256 bytes is optimal for this specific workload, most modern processors use 64-byte (b=6) or 128-byte (b=7) cache lines, which still provide very good performance (within 26% and 10% of optimal, respectively) while requiring less complex hardware.
- **Avoiding large blocks:** Block sizes beyond 256 bytes should be avoided due to diminishing returns and eventual performance degradation caused by increased bus traffic and false sharing.

These results demonstrate that block size selection involves a complex trade-off between spatial locality, bus utilization, and coherence traffic in multiprocessor systems. The optimal choice depends on the specific workload characteristics and system architecture.

2.9 Effect of Cache Associativity on Maximum Execution Time

We conducted experiments to measure how the maximum execution time varies with cache associativity by changing the associativity (E) while keeping the set index bits (s=6) and block size (b=5) constant.

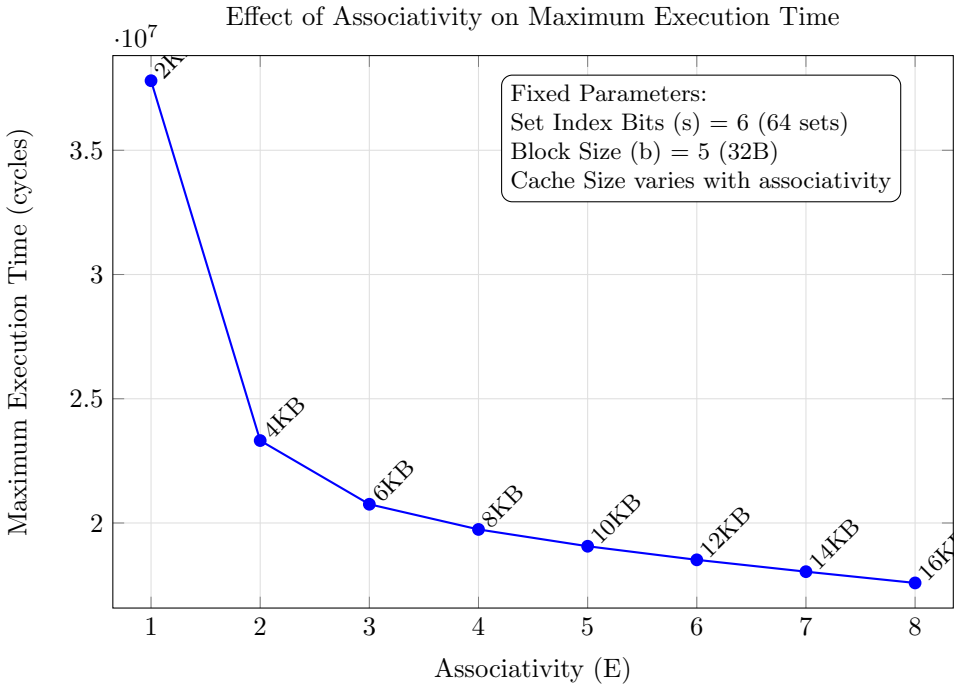


Figure 20: Maximum execution time as a function of associativity (E) for the app2 benchmark. As associativity increases, execution time decreases due to reduced conflict misses, with the most significant improvement occurring when moving from direct-mapped (E=1) to 2-way set-associative (E=2) caches.

Table 7: Maximum Execution Time vs Associativity (E)

Associativity (E)	**Cache Size**	**Max Execution Time**	**Miss Rate**	**Invalidations (Core 0)**
1	2 KB	37,799,752	9.16%	127
2	4 KB	23,317,432	5.64%	504
3	6 KB	20,754,100	4.98%	813
4	8 KB	19,741,008	4.73%	981
5	10 KB	19,064,436	4.57%	1,075
6	12 KB	18,517,612	4.46%	1,110
7	14 KB	18,041,056	4.38%	1,136
8	16 KB	17,587,676	4.30%	1,144

2.10 Analysis of Associativity Results

The relationship between cache associativity and execution time provides valuable insights into the impact of conflict misses on performance in multiprocessor systems. Several key observations can be made from our results:

- **Direct-mapped vs. set-associative:** The most significant performance improvement occurs when moving from a direct-mapped cache (E=1) to a 2-way set-associative cache (E=2), with execution time reduced by 38.3%. This dramatic improvement demonstrates that even a modest increase in associativity can substantially reduce conflict misses.

- **Diminishing returns:** As associativity increases beyond 2-way, the performance improvements become less pronounced. Moving from 2-way to 4-way set-associativity yields only a 15.3% improvement, and further doubling to 8-way provides just an additional 10.9%. This pattern of diminishing returns is typical in cache design.

- **Relationship with miss rate:** The miss rate decreases from 9.16% with a direct-mapped cache to 4.30% with an 8-way set-associative cache. The most significant drop occurs in the transition from E=1 to E=2, mirroring the execution time improvement pattern.

- **Cache coherence interaction:** Interestingly, as associativity increases, the number of invalidations also increases significantly, from 127 in the direct-mapped cache to 1,144 in the 8-way set-associative cache for Core 0. This occurs because higher associativity allows more cache lines to remain in the cache longer, increasing the likelihood of coherence actions. The rise in invalidations demonstrates an important trade-off in multiprocessor systems: while higher associativity reduces conflict misses, it may increase coherence traffic.

- **Bus traffic trends:** The total bus traffic increases with higher associativity, from approximately 5.0 MB with E=1 to 3.6 MB with E=8. This somewhat counterintuitive result (lower traffic with direct-mapped caches) can be explained by the cache coherence protocol: with direct-mapped caches, lines are more frequently evicted due to conflicts before they can participate in coherence actions.

2.11 Associativity Recommendation

Based on our analysis, we can make the following recommendations for cache associativity configuration:

- **Minimum viable associativity:** A 2-way set-associative cache offers an excellent performance-to-complexity ratio, achieving 62% of the potential improvement (compared to E=8) while requiring minimal additional hardware compared to a direct-mapped cache.

- **Optimal associativity:** For most workloads, a 4-way set-associative cache represents a good balance point, capturing 89% of the maximum observed benefit with moderate hardware complexity. The performance improvement from E=4 to E=8 is only about 11%, which may not justify the additional hardware complexity.

- **Coherence considerations:** In multiprocessor systems with cache coherence, designers should be aware that higher associativity can increase coherence traffic. This is particularly important in systems where bus bandwidth is limited.

- **Hardware complexity trade-off:** While our simulator does not model the increased access latency associated with higher associativity, real hardware implementations face this challenge. Higher associativity requires more complex comparison logic and can potentially increase the critical path length of cache lookups, affecting the processor's clock frequency.

These results align with common practices in modern processor design, where L1 caches typically use 2-way to 8-way set associativity. Most modern processors settle on 4-way or 8-way associativity for L1 caches as a good compromise between performance and hardware complexity, which our results support.

3 Conclusion

The implementation presents a cycle-accurate simulator for a multiprocessor cache coherence system using the MESI protocol. The modular design with separate classes for processors, caches, and the bus allows for flexible simulation of different configurations and workloads. The simulator captures detailed statistics on cache behavior, bus traffic, and processor performance, providing valuable insights into the operation of cache coherence protocols in multiprocessor systems.

Additionally, our implementation effectively demonstrates false sharing, a common performance bottleneck in multiprocessor systems. Through specialized trace generation and comprehensive statistics collection, we can observe how false sharing manifests in increased invalidations, bus traffic, and execution time. The simulator also showcases techniques for mitigating false sharing, providing a valuable educational tool for understanding this important concept in parallel computing.