MESI Cache Coherence Protocol Implementation

Arinjay Singhal Vihaan Luhariwala

April 30, 2025

GitHub Repository

Abstract

This document details the implementation of a cache coherence simulator using the MESI protocol. The simulator models multiple processor cores with private L1 caches communicating over a shared bus, implementing coherence protocols to maintain data consistency across the system.

Contents

1 Implementation

1.1 Core Classes and Their Structures

1.1.1 Cache Class

The Cache class implements the L1 data cache with:

- **Cache configuration:** Configurable parameters for sets, ways, and block size to support different cache architectures and optimization strategies
- **2D vector structure:** Efficient organization of cache data using a two-dimensional vector, enabling O(1) access time for cache operations
- **LRU counters:** Implementation of Least Recently Used replacement policy to optimize cache utilization and hit rates
- **Statistics tracking:** Comprehensive monitoring of cache performance metrics including hits, misses, and writebacks

1.1.2 CacheLine Structure

Each cache line maintains:

- **MESI state and dirty bits:** Current coherence state (Modified, Exclusive, Shared, Invalid) and write-back status tracking
- **Tag bits:** Address identification for cache line matching and validation
- **Data block storage:** Actual cached memory content with configurable block size
- **LRU counter:** Dynamic tracking of access patterns for replacement decisions

1.1.3 Bus Class

The Bus class manages:

- **Bus configuration:** Settings for bus width, transaction types, and connected cores
- **Bus state tracking:** Real-time monitoring of bus availability and current transaction status
- **Transaction management:** Handling of read, write, and coherence messages between caches
- **Coherence statistics:** Collection of bus utilization and protocol overhead metrics

2 Data Structures and Assumptions

2.1 Core Data Structures

2.1.1 Cache Line Structure

- **tag:** Address tag bits for cache line identification and matching operations
- **state:** Current MESI protocol state indicating line validity and sharing status
- **data:** Actual memory content stored in the cache line
- **lru_counter:** Timestamp-based counter for replacement policy implementation

2.1.2 Bus Transaction Structure

- **type:** Transaction classification (Process/Broadcast) determining handling protocol
- **address:** Target memory address for the current operation
- **requesting_core:** Identifier of the core initiating the transaction
- **owning_core:** Identifier of the core currently holding the data
- **data:** Memory content being transferred across the bus

2.2 Key Assumptions

1. Execution cycles: A cycle is counted as execution cycle of a core in following cases:
   - Any hit operation is performed which does not require the bus.
   - Bus is involved in executing the operations due to the corresponding core's instruction.
2. Idle cycles: A cycle is counted as idle cycle of a core if the core wants to access the bus but it is involved in any other core's operation.
3. If multiple cores want to access the bus at the same time, then the bus is assigned to the core with the lowest id.
4. Memory writeback happens in following cases:
   - Eviction of Modified lines: due to capacity constrains, the cache line is evicted and written back to the memory. This is counted in execution cycles of the core that caused the eviction.
   - Bus request of Modified lines: When a cache line is in modified state and other core requests the data, then the data is written back to the memory and this is counted in execution cycles of the core that caused the writeback. The core which writes back to memory is not halted in this case unless it requires the bus (simple read hits can occur in this case).
5. Reading/Writing from memory takes 100 cycles. So if a single read operation is performed, it takes 100 cycles to complete and an additional cycle to do the read hit. So total 101 cycles.
6. Cache to Cache data transfer takes 2N cycles where N is the number of words in the cache line. The bus is busy in this case. These cycles are counted in execution cycles of the core that caused the data transfer. The core that supplies the data is not stalled (and can process read hits), if it requests bus access then only it is stalled.
7. Snooping and initiating bus transaction happens in the same cycle as the miss is detected. Bus is busy until the transaction is completed.
8. The bus can handle only one transaction at a time. So Snooping signals cannot pass through the bus if it is busy.
9. Bus transaction signals :
   - BusRd: A core requests the bus for read operation.
   - BusRdX: Read with intent to Modify, this is used when the core wants to read the data and then write to it.
   - BusUpgr: If the core has a value in Shared state, and it wants to write over it, then it sends BusUpgr signal.
10. Invalidation counts: Total number of instructions passed by a core which cause invalidation of any cache line in any other core.
11. If a instruction encounters a cache miss, the miss count is incremented. The miss is counted only once per instruction.

3 Function Flowcharts

3.1 Main Program Flow

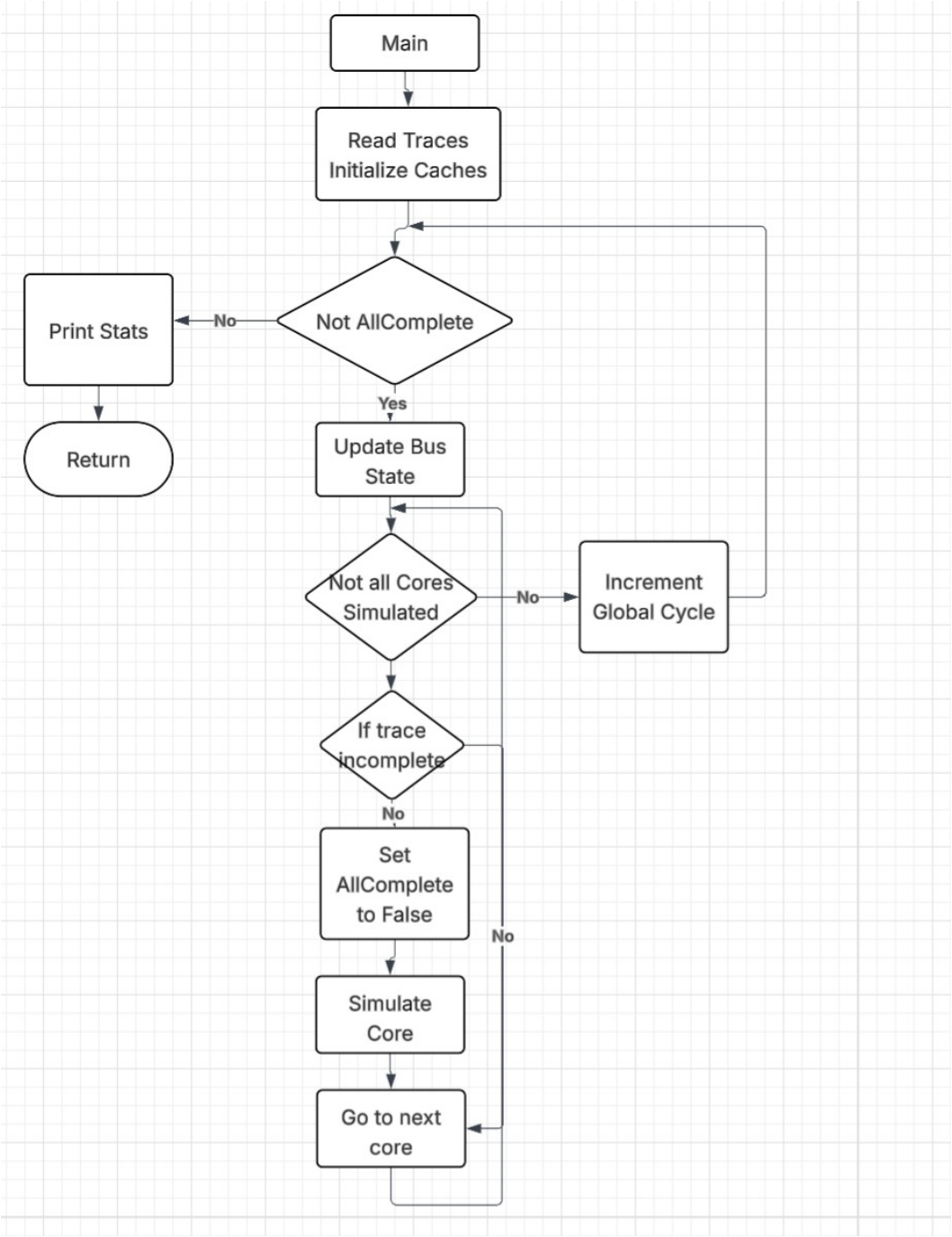

Figure 1: Main program execution flow

3.2 Bus Transaction Processing

Figure 2: Bus transaction processing flow

3.3 Cache Operations

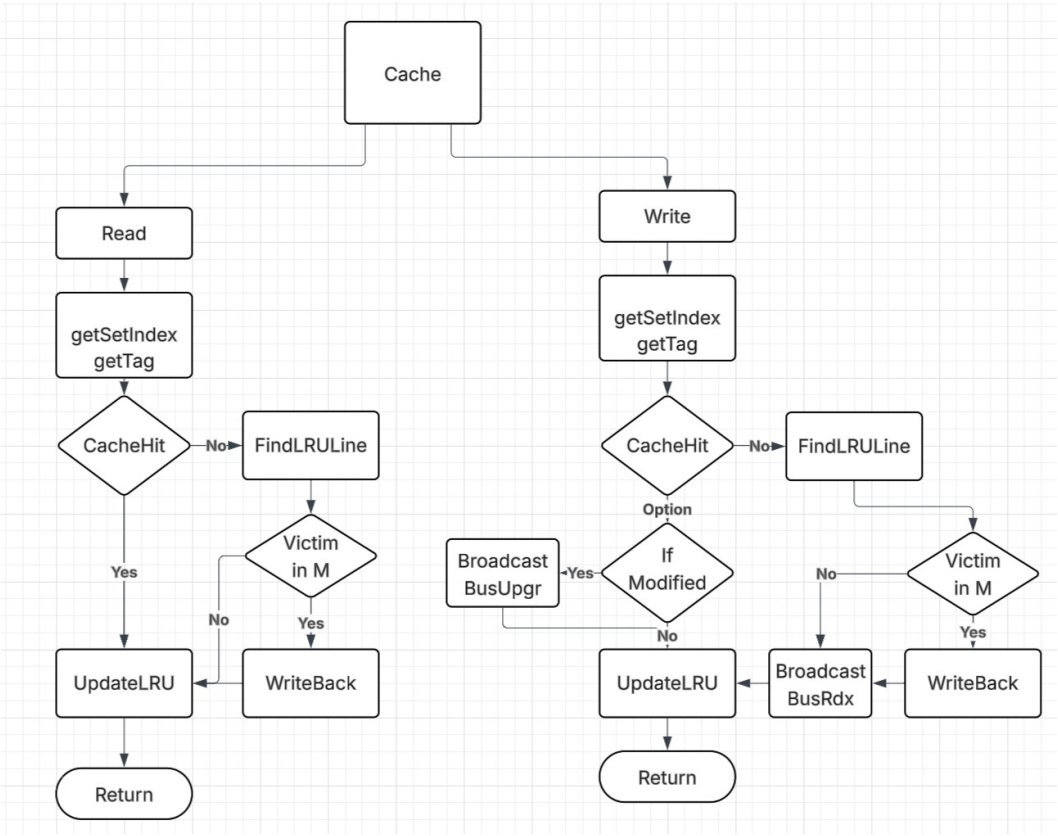

Figure 3: Cache read and write operations

3.4 Process Bus Transaction

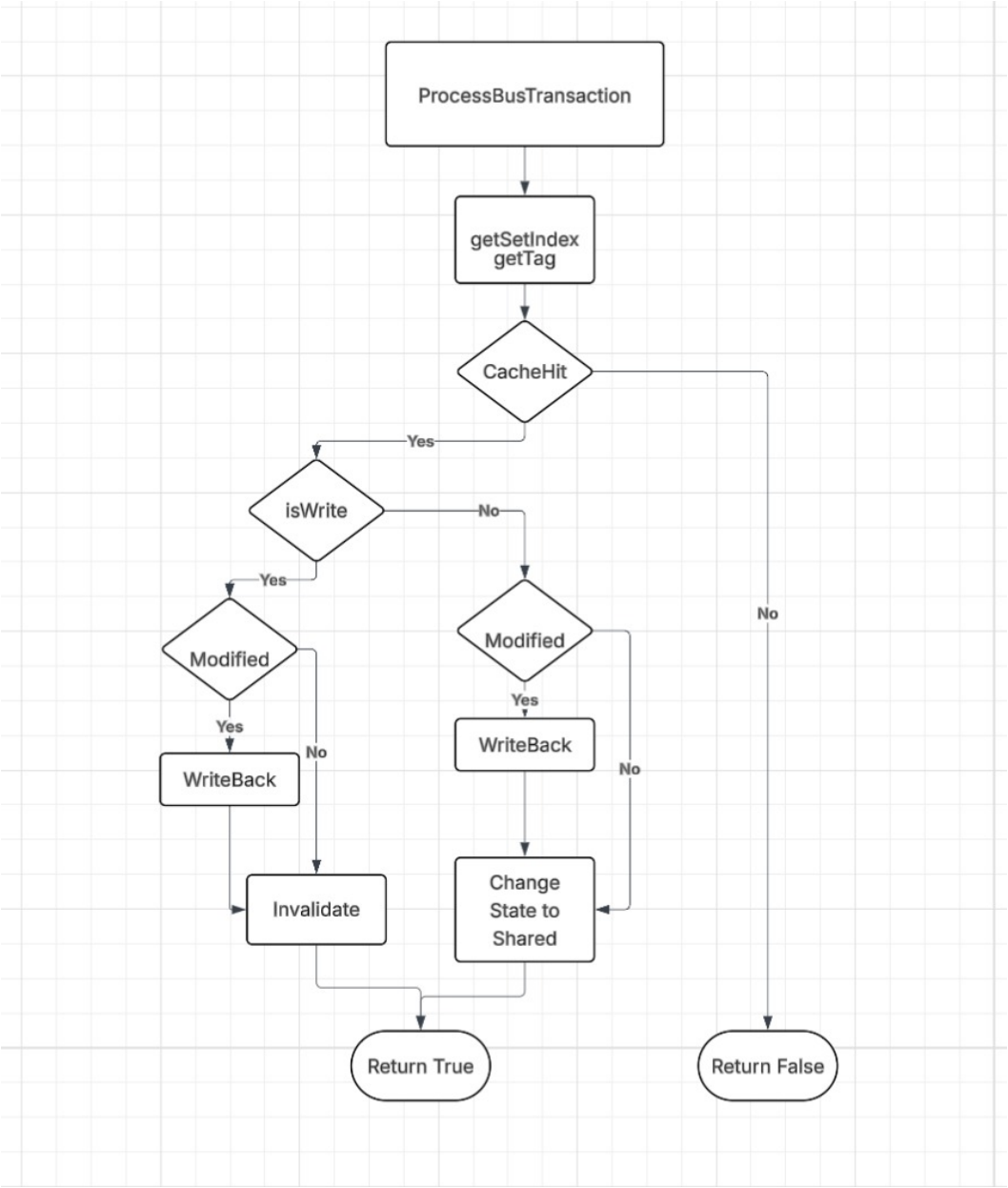

Figure 4: Process bus transaction flow

4 Protocol Implementation

4.1 MESI State Transitions

The MESI protocol implements state transitions based on bus transactions and local operations:

- **Modified → Shared/Invalid:** Occurs when other cores request read access or line eviction
- **Exclusive → Modified/Shared/Invalid:** Transitions on write hits, read requests, or evictions
- **Shared → Modified/Invalid:** Changes on write operations or invalidation requests
- **Invalid → Modified/Exclusive/Shared:** Results from read/write misses with varying sharing states

5 Performance Evaluation

5.1 Benchmark Results

Performance metrics include:

- **Cache hit/miss rates:** Measurement of cache efficiency and access pattern optimization
- **Bus utilization:** Analysis of bus bandwidth usage and transaction patterns
- **Coherence overhead:** Impact of coherence protocol on system performance
- **Memory access patterns:** Study of spatial and temporal locality in applications

6 Trace Analysis

Note: We have implemented LRU policy in case of bus access races. This makes our model deterministic. So the rubric where we have to generate multiple runs for same parameters are not applicable to us.

6.1 False Sharing Analysis

False sharing occurs when multiple processors access different data items that happen to be stored in the same cache line. This analysis compares the performance impact of false sharing with different block sizes. Traces are available in falsesharing folder.

6.1.1 Block Size = 5 (32 bytes)

- Cache Miss Rate: 66.67% across all cores
- Execution Cycles:
  - Core 0: 303 cycles
  - Core 1: 403 cycles
  - Core 2: 403 cycles
  - Core 3: 403 cycles
- Idle Cycles:
  - Core 0: 199 cycles
  - Core 1: 300 cycles
  - Core 2: 899 cycles
  - Core 3: 1100 cycles

6.1.2 Block Size = 2 (4 bytes)

- Cache Miss Rate: 33.33% across all cores
- Execution Cycles:
  - Core 0: 103 cycles
  - Core 1: 103 cycles
  - Core 2: 103 cycles
  - Core 3: 103 cycles
- Idle Cycles:
  - Core 0: 0 cycles
  - Core 1: 100 cycles
  - Core 2: 200 cycles
  - Core 3: 300 cycles

6.1.3 Analysis of False Sharing

The results demonstrate the impact of false sharing on system performance:

- **High Miss Rates:** The 66.67% miss rate indicates frequent cache line invalidations due to false sharing
- **Bus Contention:** All bus transactions are BusRdX, showing that writes are causing coherence traffic
- **Performance Impact:** The high number of writebacks and invalidations indicates significant overhead from false sharing
- **Block Size Effect:** Larger block sizes (b=5) exacerbate false sharing as more data items share the same cache line

6.2 LRU testing

The test case has the following parameters:

- Set Index Bits (s) = 0
- Associativity (E) = 3
- Block Size (b) = 2 (4 bytes)

This configuration results in a single set with 3 possible ways. The test sequence demonstrates the LRU replacement policy:

1. Initial compulsory read misses for addresses 0x0, 0x4, and 0x8 establish the initial LRU order: 0x0, 0x4, 0x8
2. A subsequent read to 0x4 updates the LRU order to: 0x0, 0x8, 0x4
3. Two write operations to 0xC and 0x10 trigger evictions:
    - First write to 0xC evicts 0x0, resulting in LRU order: 0x8, 0x4, 0xC
    - Second write to 0x10 evicts 0x8, completing the sequence

This behavior was verified through debug mode execution, confirming the correct LRU order and eviction sequence.

6.2.1 Data Traffic Analysis

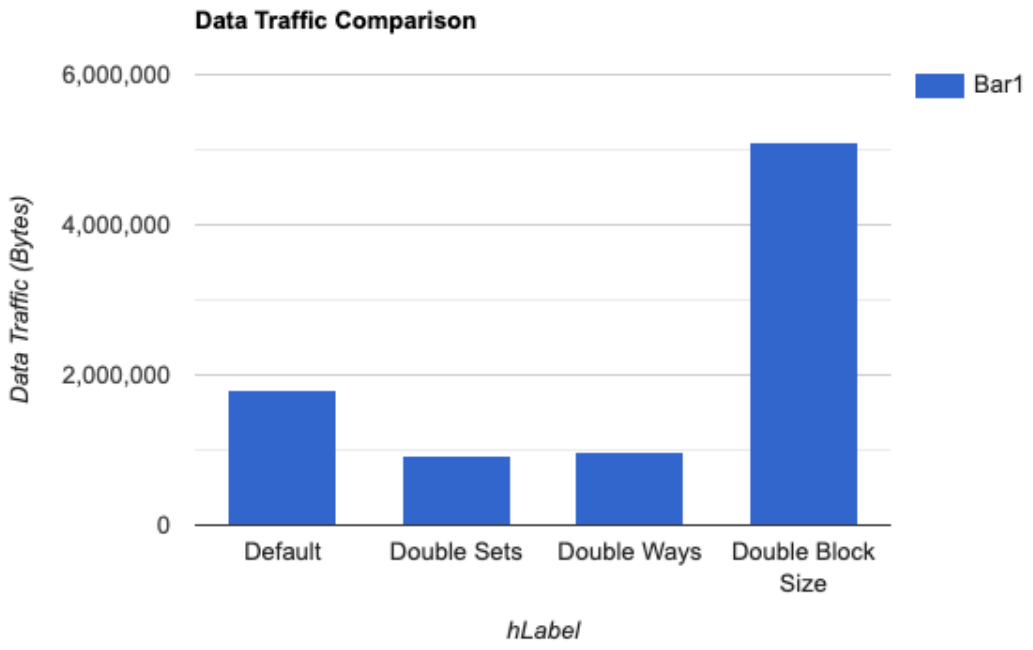

Figure 5: Data Traffic Comparison for Different Cache Configurations

The data traffic analysis reveals significant variations across different cache configurations:

- **Block Size Impact:** Doubling block size causes the highest data traffic due to larger memory transfers per miss. While this can potentially improve spatial locality, it significantly increases bandwidth usage.
- **Cache Organization:** Doubling sets or ways reduces data traffic, indicating fewer memory accesses and better cache efficiency. This suggests that increasing cache capacity through either more sets or higher associativity can effectively reduce memory bandwidth requirements.
- **Trade-offs:** While larger blocks can increase spatial locality by prefetching nearby data, they also lead to increased bandwidth usage. This highlights the importance of balancing spatial locality benefits against bandwidth constraints in cache design.

6.2.2 Evictions Analysis

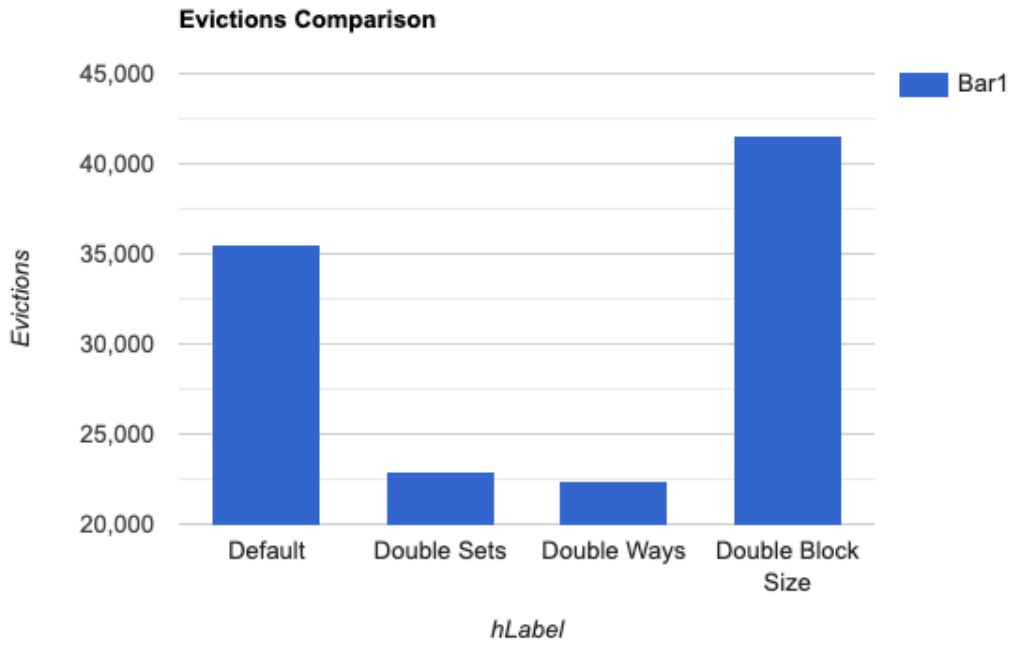

Figure 6: Cache Evictions Comparison for Different Cache Configurations

The evictions analysis demonstrates how different cache parameters affect the frequency of cache line replacements:

- **Cache Capacity Impact:** Double sets and double ways result in the fewest cache evictions, thanks to greater capacity and associativity. This demonstrates how increased cache space reduces conflict and capacity misses.
- **Block Size Effect:** Double block size leads to the most evictions, as fewer larger blocks fit in the cache. This illustrates the trade-off between block size and effective cache utilization.
- **Baseline Performance:** The default configuration shows moderate evictions, reflecting limited cache flexibility. This serves as a reference point for evaluating optimization strategies.

6.2.3 Execution Cycles Analysis

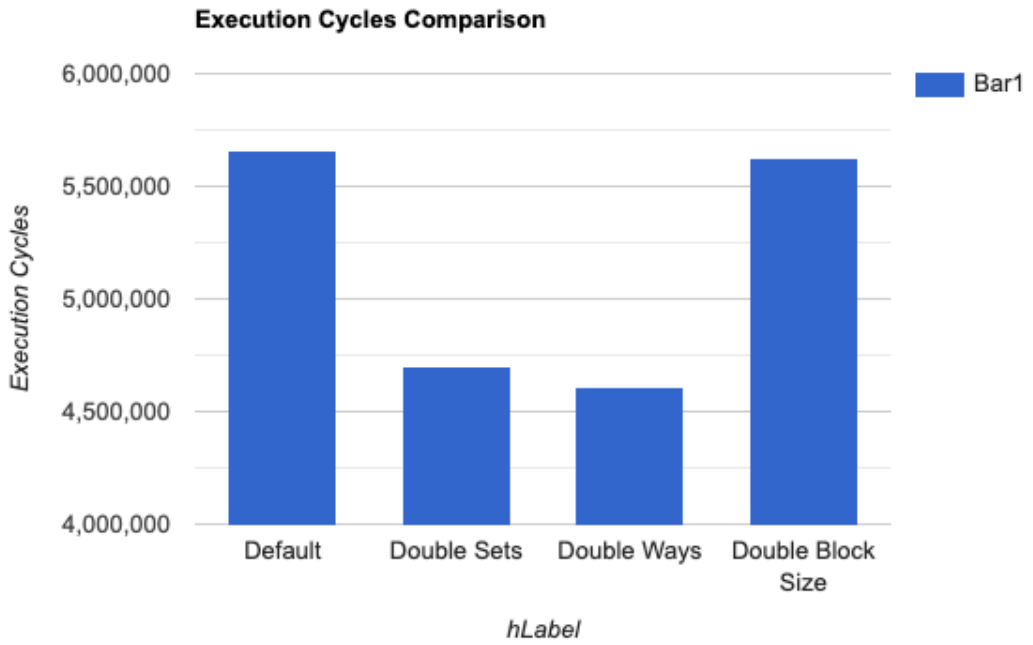

Figure 7: Execution Cycles Comparison for Different Cache Configurations

The execution cycles analysis reveals the impact of cache configurations on overall performance:

- **Associativity Impact:** Doubling the number of ways yields the lowest execution cycles, improving performance. This suggests that increased associativity effectively reduces conflict misses and improves cache utilization.
- **Set Size Effect:** Double sets also improve performance, but slightly less than double ways. This indicates that while increasing the number of sets reduces conflicts, the benefit is not as pronounced as increasing associativity.
- **Block Size Trade-off:** Double block size performs similarly to default, suggesting performance gains from spatial locality are offset by higher miss costs. This demonstrates the complex balance between exploiting spatial locality and managing cache space efficiently.

6.2.4 Idle Cycles Analysis

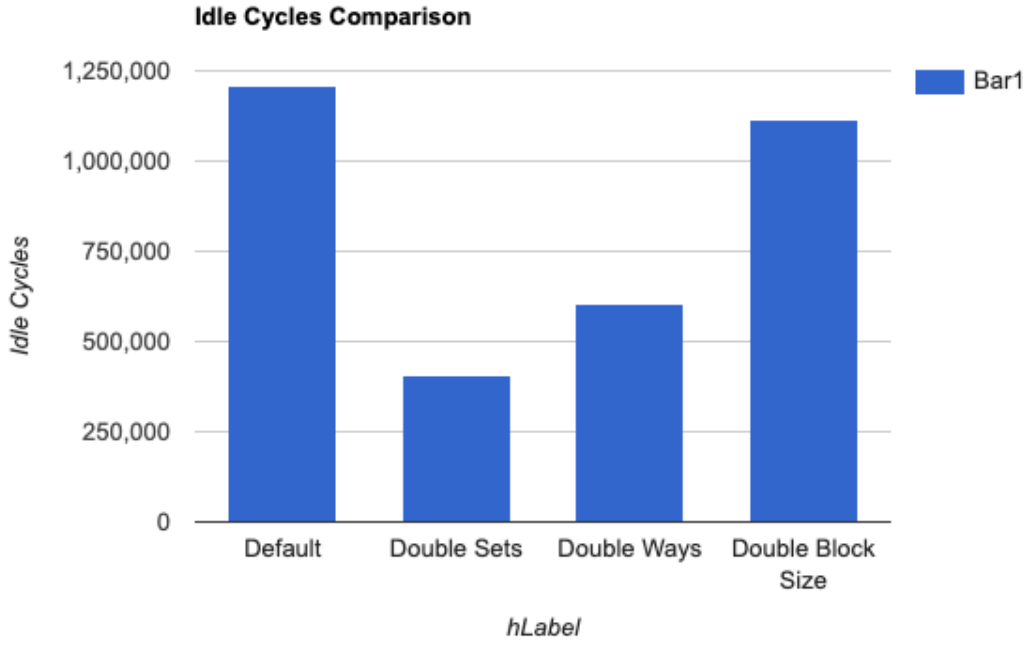

Figure 8: Idle Cycles Comparison for Different Cache Configurations

The idle cycles analysis provides insights into memory-related stalls across different configurations:

- **Set Size Efficiency:** Double sets show the least idle time, indicating fewer memory stalls. This suggests that increasing the number of sets leads to better distribution of cache lines and reduced contention for memory access.
- **Associativity Benefits:** Double ways also reduce idle cycles, due to better hit rates. The increased associativity helps avoid conflict misses, resulting in fewer memory access delays.
- **Block Size Impact:** Double block size increases idle time, as larger misses take longer to fetch. This highlights how larger block sizes can lead to increased memory access latency despite potential spatial locality benefits.
- **Default Limitations:** The default configuration shows the most idle cycles, indicating frequent memory-related stalls. This baseline performance suggests significant room for improvement through cache parameter optimization.

6.2.5 Invalidations Analysis

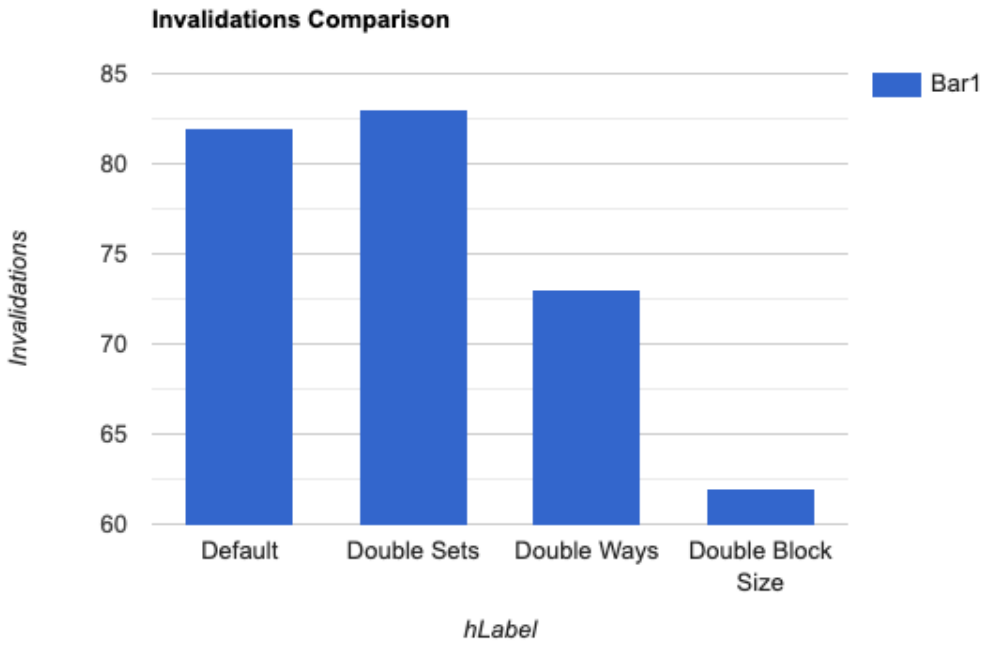

Figure 9: Cache Invalidations Comparison for Different Cache Configurations

The invalidations analysis reveals the coherence behavior in the multicore environment:

- **Block Size Effect:** Double block size results in the fewest invalidations, possibly due to fewer, larger blocks being shared. This suggests that larger blocks may reduce coherence traffic in certain workloads, despite their other performance trade-offs.
- **Set Size Impact:** Double sets lead to the most invalidations, likely due to more active cache blocks in a multicore environment. While more sets improve overall cache utilization, they may increase coherence overhead in shared memory scenarios.
- **Associativity Influence:** Double ways shows moderate improvement in invalidations compared to the default, suggesting that increased associativity can help manage coherence traffic while maintaining good cache utilization.

6.2.6 Miss Rate Analysis

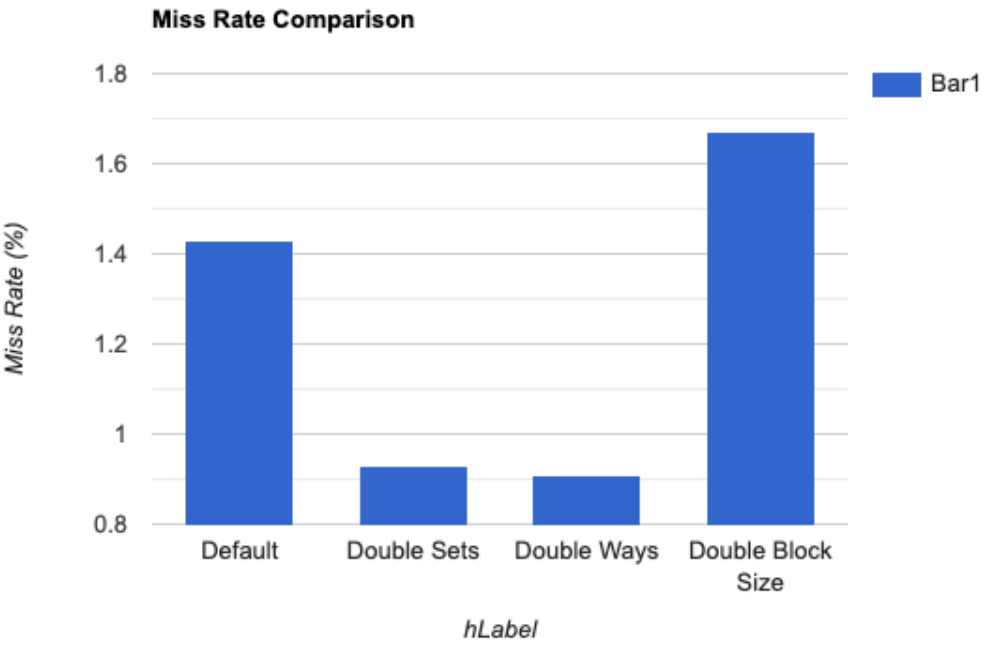

Figure 10: Cache Miss Rate Comparison for Different Cache Configurations

The miss rate analysis demonstrates the effectiveness of different cache organizations:

- **Cache Organization Benefits:** Double sets and double ways significantly reduce miss rate, improving cache hit efficiency. This demonstrates how both increased associativity and more sets can effectively reduce conflict misses and improve overall cache performance.
- **Block Size Impact:** Double block size increases miss rate, as fewer blocks reduce flexibility and increase conflict/capacity misses. This higher miss rate suggests that the potential benefits of spatial locality are outweighed by the reduced number of cache blocks available.
- **Baseline Performance:** Default configuration has a moderate miss rate, with room for improvement. This indicates that simple modifications to cache parameters can yield significant performance benefits.

6.2.7 Total Program Cycles Analysis

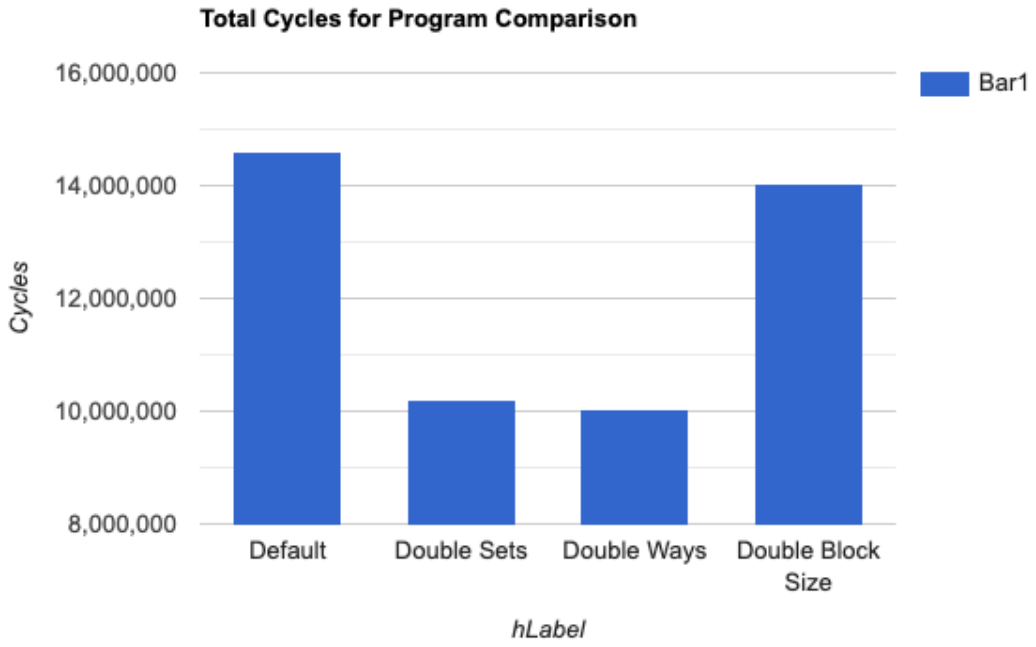

Figure 11: Total Program Cycles Comparison for Different Cache Configurations

The total program cycles analysis provides a comprehensive view of overall system performance:

- **Set Size Impact:** Double Sets reduce total cycles by about 30% (from 14.5M to 10.2M), offering a significant performance boost through reduced conflict misses. This substantial improvement demonstrates the effectiveness of increasing cache capacity through additional sets.
- **Associativity Benefits:** Double Ways perform slightly better than double sets (10M cycles), benefiting from higher associativity that improves cache efficiency. This suggests that increased associativity is particularly effective at reducing conflict misses in this workload.
- **Block Size Effect:** Double Block Size shows minimal improvement (14M cycles), suggesting limited benefit from larger blocks due to poor spatial locality in the workload. This indicates that simply increasing block size may not be effective without considering the application's memory access patterns.
- **Default Limitations:** Default configuration has the highest total cycles (14.5M), reflecting inefficient cache behavior and frequent memory stalls. This baseline performance emphasizes the potential for significant improvements through cache parameter optimization.

6.2.8 Write Backs Analysis

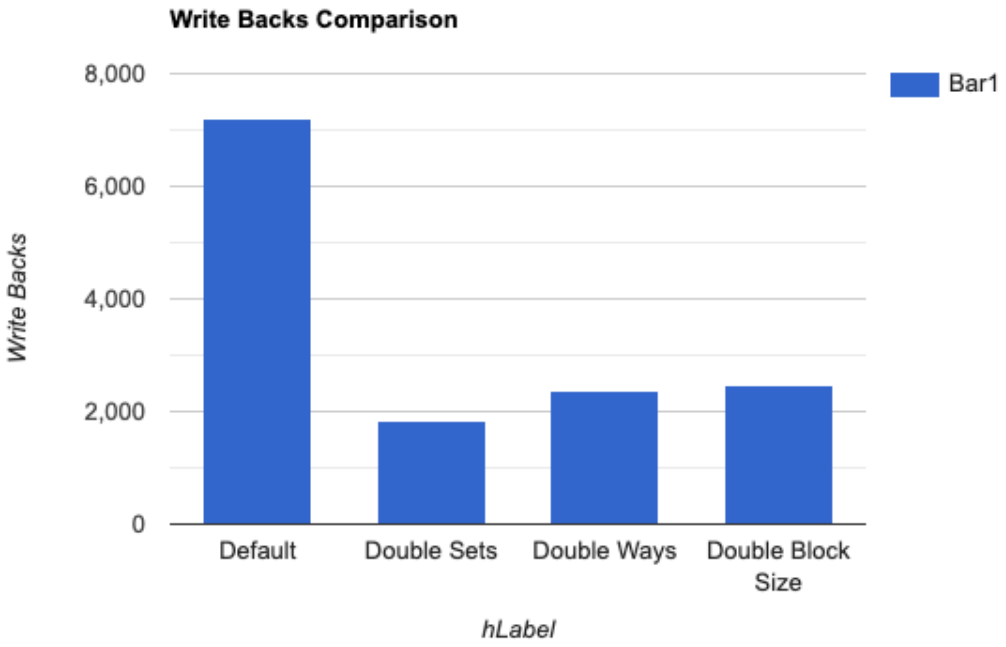

Figure 12: Write Backs Comparison for Different Cache Configurations

The write backs analysis reveals how different cache configurations affect memory write traffic:

- **Set Size Efficiency:** Double Sets reduce write backs drastically (from 7,300 to 1,800), showing the best performance by minimizing evictions of dirty blocks. This significant reduction demonstrates how increased set count can effectively reduce capacity-related writebacks.
- **Associativity Impact:** Double Ways achieve a 67% reduction (to 2,400 write backs), effectively lowering write-backs through improved associativity, but not as efficiently as doubling sets. This suggests that while higher associativity helps, set organization has a more significant impact on write-back behavior.
- **Block Size Effect:** Double Block Size results in 2,500 write backs, slightly higher than double ways, as larger blocks group writes better but don't reduce eviction pressure as much. This shows the trade-off between spatial write grouping and cache space utilization.
- **Default Limitations:** Default configuration has the highest write backs (7,300), caused by frequent eviction of modified lines in a limited cache space. This indicates significant room for improvement in write-back efficiency through cache parameter optimization.

7 Conclusion

The implementation demonstrates a functional MESI cache coherence protocol with:

- **Efficient state management:** Optimized handling of cache line states and transitions
- **Proper coherence maintenance:** Reliable data consistency across multiple caches
- **Scalable multi-core support:** Architecture supporting varying numbers of cores
- **Reliable performance metrics:** Comprehensive monitoring and analysis capabilities