CS4223 Multi-Core Architectures

A Quad-core Cache Coherence Simulator for MESI, Dragon and MESIF Protocols

Wang Hongxiang(A0279535N)
Sun Jingjing(A0286172Y)

17 November 2023

Contents

1 Introduction

In this assignment, we implemented a simulator for three cache coherence protocols: MESI, Dragon and MESIF, and analyzing their performance.

Based on our own programming experiences, we used Python as our coding language on the macOS platform. We successfully implemented the MESI, Dragon, and MESIF protocols within our simulation environment.

2 Simulator Design and Implementation

Our simulator is an object-oriented design, with components(bus, cache and core) in the system modelled by their own class.

In function ‘main.py’, we parse the five input arguments:

- “protocol” is either MESI, Dragon or MESIF
- “input_file” is the input benchmark name (e.g., bodytrack)
- “cache_size”: cache size in bytes, multiples of words(each word is 4 bytes)
- “associativity”: associativity of the cache
- “block_size”: block size in bytes, multiples of words(each word is 4 bytes)

For example, to read bodytrack trace files and execute MESI cache coherence protocol with each core containing 1K direct-mapped cache and 16 byte block size, the command will be “python3 main.py MESI bodytrack 1024 1 16”

According to parse the “protocol”, the right Simulator will be invoked to perform the simulation. The Simulator for each protocol is responsible for initialing parameters of cache size, associativity and block size, encoding every state, transitions and bus transitions.

2.1 Processor Cores

Processor cores are modelled by the **Core** class associate with its own cache, which has 2 states: free and busy. When a **Core** is free, it is allowed to initiate a new transaction on the shared bus. A **Core** in the busy state will be free when it finishes its current action. However, when a **Core** is busy, it is not allowed to initiate a new transaction on the shared bus, but it can snoop on the shared bus and respond to the bus activity. A **Core** becomes busy if it performs computation, and when start a load or store instruction.

Additionally, the **Core** class is used to account the information of execution cycle, Compute time, Idle time and so on.

2.2 Cache

The L1 data caches are modelled by **Cache** class, which is an implementation of write-back, write-allocate policy and LRU replacement policy.

A **Cache** is stateless, but associated with some parameters, such as block size, cache size and the number of set-associative. We use a **Cacheline** class to represent a cache block and it has a state that is managed by different **Simulator.**

In detail, the attributes of Cacheline are**:** State, LastUsed, BlockNumber, ValidFrom. The 'lastused' value for each **Cacheline** object is used for LRU replacement policy. To avoid a latency penalty in filling a new cache line when an evicted block has to be written back to memory, we assume the L1 caches employ Line Fill Buffers (LFB) to store incoming data in-transit from the bus.

2.3 Bus

The **Bus** class in our simulator maintains statistics on bus traffic and transaction composition. Initially, we considered an atomic bus, allowing only one cache to communicate at a time. While easy to implement, this approach significantly reduces bandwidth as the bus idles during transaction responses, serializing memory requests across caches.

To enhance performance, we chose a split transaction bus design. Transactions are divided into requests and responses, enabling pipelining and out-of-order completion of requests. The request order dictates the system's total order. We allow one request per cycle on the request bus and assume the response bus has ample bandwidth for data.

Each cycle, the Simulator's bus arbiter permits only one cache to make a new request, prioritizing the cache with the oldest pending request to prevent starvation. When a new request is placed, the snooping caches update their state, and the designated cache/memory prepares the response.
To avoid conflicts over a pending block, our Runner's request table tracks outstanding requests (up to four). Caches requesting a pending block are stalled until the corresponding response completes.

Our bus clients can service incoming transactions while queuing to issue requests, preventing deadlocks. However, to avoid livelock, particularly under the MESI protocol when two processors write to the same cache line, our bus ensures a cache completes one operation before another request for the same block is granted.

Lastly, our memory controller uses a write-back buffer for invalidated or evicted blocks, not checking the buffer against new requests. If cache-to-cache transfer is unavailable, a new request must wait until the write-back completes, which can take 100 cycles.

2.4 Simulator

The Simulator is designed to emulate cache coherence protocols within a system comprising one common Bus, four Core instances, and four Cache instances.

To avoid conflicting requests, the Simulator monitors cache blocks currently engaged in transactions, whether between memory and cache or among caches themselves. It will delay a Core's write operations and read/write requests if they target a block that is already part of an ongoing transaction.

There are 3 subtypes of **Simulator** implemented:
- **MESIsim** Simulates the classic no-intervention MESI protocol.
- **Dragonsim**: Simulates the Dragon protocol.
- **MESIFsim**: Simulates the MESIF protocol.

Each **Simulator** implements the four scenarios: simulate_read_miss, simulate_read_hit, simulate_write_miss, and simulate_ write _hit.

During each cycle, the Simulator prioritizes Cores that are in a free state for processing. To prevent any Core from being deprived of bus access, the Simulator arranges the Cores according to the last cycle in which they accessed the Bus. Cores that have accessed the Bus more recently are assigned a lower priority. In cases where two Cores have accessed the Bus at the same time in the past, priority is determined by their ID numbers; the Core with the smaller ID receives higher priority.

After sorting, the Simulator will process each Core's next trace (if execution is possible). A **Core**'s trace might be stalled due to serval reasons: not being granted bus access, or the requested block is active in a pending request.

A block's availability time in the memory is also recorded. If availability is INF, the block's latest copy is guaranteed to be stored in some Cache of some core.

3 Cache Coherence Protocols

3.1 MESI Protocol

The MESI protocol is an invalidation-based cache coherence model that encompasses four states for a cache line: Modified (M), Exclusive (E), Shared (S), and Invalid (I). In this system, when a cache miss occurs, the required block is retrieved directly from the main memory, since cache-to-cache transfers are not permitted. Consequently, if a cache needs to access a block that is currently in the Modified state in another cache, it must wait for that cache to write the block back to the

main memory. This process incurs an additional latency penalty due to the write-back operation and the subsequent memory access needed to retrieve the block.

For any given pair of caches, the permitted states of a given cache line are as follows, and the MESI state transition diagram is shown in figure1 :

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

When the block is marked M (modified) or E (exclusive), the copies of the block in other Caches are marked as I (Invalid).

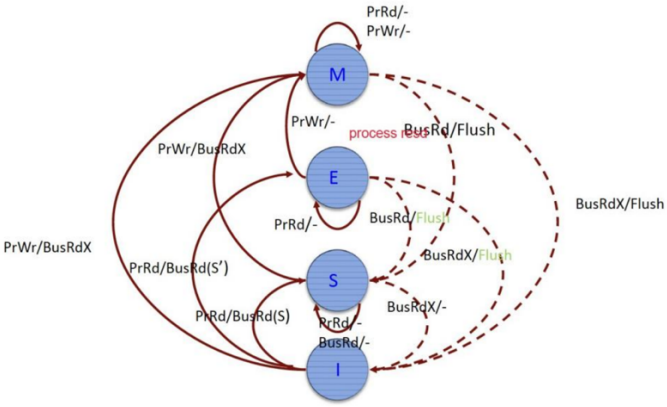


Figure1: State diagram for MESI

This protocol is implemented in the MESIsim class, and there are some main instructions:

(1) simulate_read_hit: The cache of the Core is set to last used.

simulate_read_miss: If the block is not currently being transacted and the bus is available, a BusRd operation is isseued. If the requested block is found in the Modified (M) state in another cache, the requesting cache must stall until the block has been written back to memory. The block is then retrieved from memory, and the **Core** is set to busy until the block has been filled in the **Cache**.

(2) simulate_store_hit: The block of the Core is set to last used and set the state to M.

simulate_store_miss: If the block is not currently being transacted and the bus is available, a BusRdX operation is isseued. If the requested block is found in the Modified (M) state in some other cache, the requesting cache must stall until the block has been written back to memory. The BusRdX operation will also invalidate any clean copies of the block in other caches. Subsequently, the block is retrieved from memory. The **Core** is set to busy until the block has been filled in the **Cache**.

(3) compute

Set the Core to busy for the required number of cycles.

3.2 Dragon Protocol

The Dragon protocol is an advancement over the MESI protocol, enhancing efficiency by enabling transfers between caches. It manages cache coherence with four states: Exclusive, Shared Clean, Shared Modified, and Modified. This approach reduces bus traffic and improves system performance by allowing caches to share data directly without involving the main memory as frequently.

For any given pair of caches, the permitted states of a given cache line are as follows, and the MESI state transition diagram is shown in figure2 :

	E	Sc	Sm	M
E	✗	✗	✗	✗
Sc	✗	✓	✓	✗
Sm	✗	✓	✗	✗
M	✗	✗	✗	✗

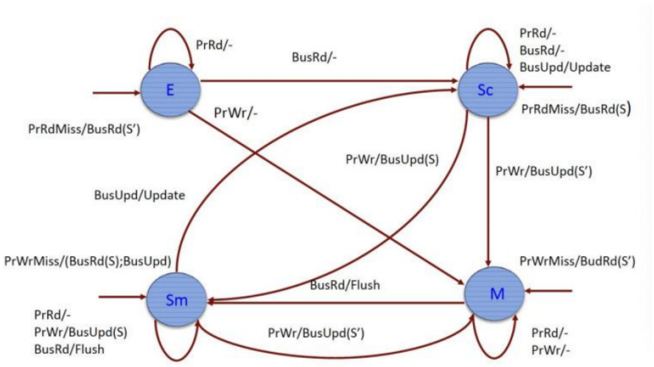


Figure2: State diagram for Dragon

This protocol is implemented in the Dragonsim class, and there are three principal instructions:

(1) **simulate_read_hit**: The cache of the Core is set to last used.
simulate_read_miss: If the block is not currently being transacted and the bus is available, processd to fetch and allocate a copy of the block from: memory if no other caches have a copy, otherwise from another cache. The **Core** is set to busy until the block has been filled in the **Cache.**

(2) **simulate_store_hit**: The Core is set to busy for 1 cycle. If the block is not in the Modified state, there would be broadcasting of the written word, which requires an additional cycles.
simulate_store_miss: If the block is not currently being transacted and the **Bus** is available, processd to fetch and allocate a copy of the block from: memory if no other caches have a copy, otherwise from another cache. The **Core** is set to busy until the block has been filled in the **Cache**. Finally, the written word might be broadcast, taking 2 cycles.

(3) **Compute**
Set the **Core** to busy for the required number of cycles.

3.3 Advanced task: MESIF invalidation-based protocol

The MESIF protocol consists of five states, **M(Modified), E(Exclusive), S(Shared), I(Invalid) and F(Forward)**. The M, E, S and I states are the same as in the MESI protocol, and the F state, which designates a specific cache as the primary source for a particular cache line in the shared state. This means that when a read request is made for a cache line, it can be directly served by the cache in the F state, bypassing the need to access the main memory. The MESIF protocol ensures that if any cache has a line in the Shared (S) state, only one other cache at most can have it in the Forward (F) state at the same time. This mechanism allows for efficient utilization of the cache and memory resources in a multicore processor environment, thereby improving overall system performance.

For any given pair of caches, the permitted states of a given cache line are as follows, and the MESIF state transition diagram is shown in figure3 :

	M	E	S	I	F
M	✗	✗	✗	✓	✗
E	✗	✗	✗	✓	✗
S	✗	✗	✓	✓	✓
I	✓	✓	✓	✓	✓
F	✗	✗	✓	✓	✗

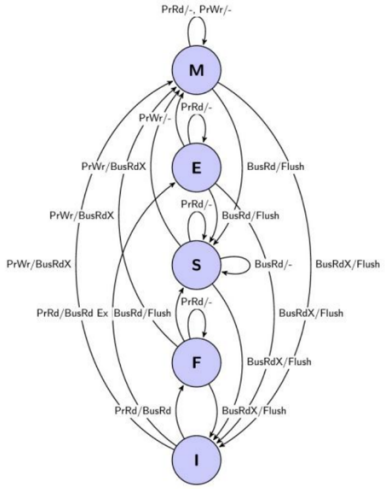


Figure3: State diagram for MESIF

This protocol is implemented in the MESIFsim class, and there are two main differences from the classical MESI implementation:

(1) **simulate_read_miss:** The block is retrieved from memory if no other cache holds a copy(after all write-backs for that block complete). If other caches have the block in Modified (M), Exclusive (E), or Forward (F) states, the block is acquired from the quickest source, either memory or the forwarding cache. Subsequently, the cache supplying the block is designated the Forward (F) state if additional shared copies are present, or the Exclusive (E) state if not.

(2) **simulate_write_miss:** The block is retrieved from memory if it is not present in any other cache. If it exists in a forwarding cache with Modified (M), Exclusive (E), or Forward (F) states, the source that can provide the block the quickest is used. Once the block is acquired, the cache

that retrieves it for modification is designated the Modified (M) state, indicating it has the intent to write to the block.

4 Results and analysis

In this assignment, there are three benchmark traces from the PARSEC benchmark: **blackscholes, bodytrack** and **fluidanimate**. The default parameters are: 4096-byte cache size, 2-way set associativity and a 16-byte block size.

A summary of programming models used in each benchmark is shown as below:

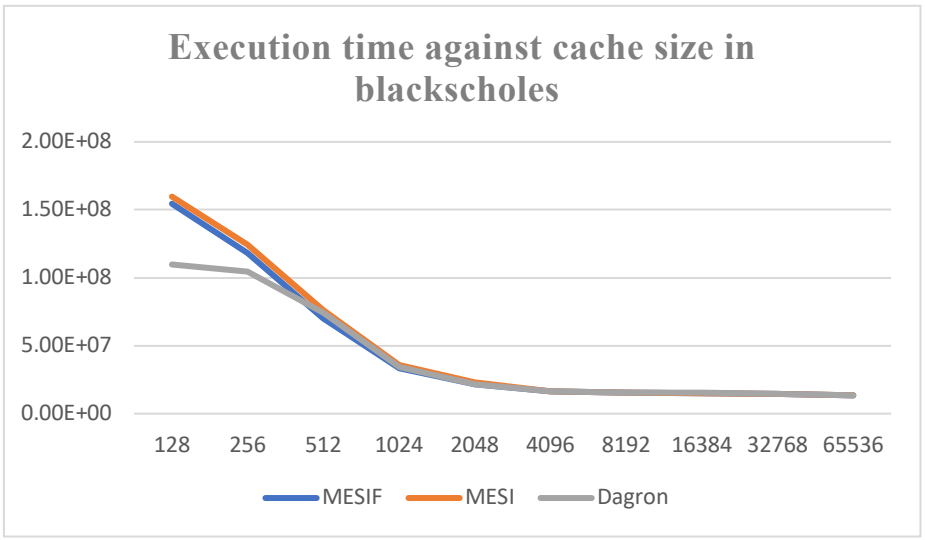
benchmark	Computation	Pthreads model	OpenMP model	TBB model	Task models
blackscholes	for (100 runs) { 1 for loop}	SPMD	omp parallel for	tbb:parallel_for	pfor
bodytrack	for (261 steps) { 5 for loops}	manual task queue	omp parallel for	tbb:pipeline tbb:parallel_for	Pipeline tasks pfor
fludanimate	for (500 steps) { 1 for loop}	SPMD		tbb:task	pfor

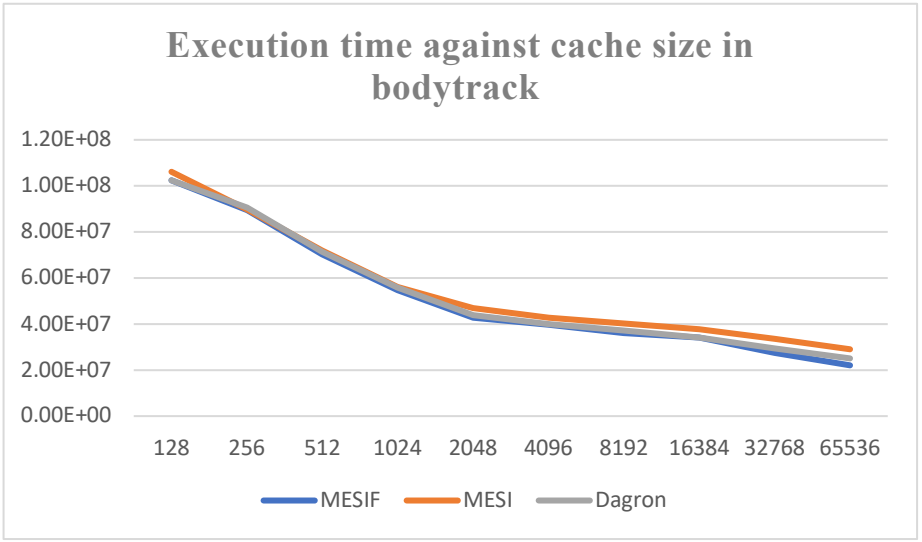
Through the experiments, we discovered that Compute time, cache miss rate and Load/Store counts are independent of the cache coherence protocol used. This means that these metrics remain constant across different protocols when running the same benchmark.

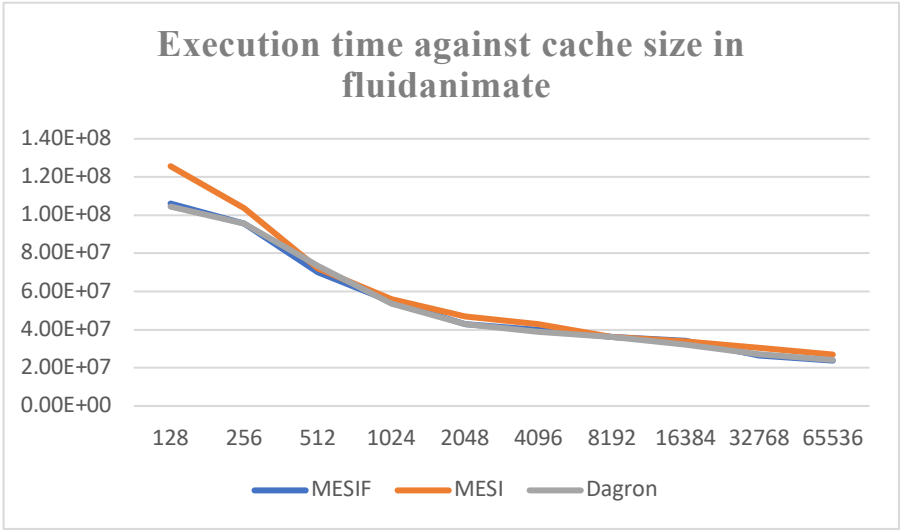
In the experiment, based on the default parameters, we tested with three parameters(cache size, associativity, and block size), each being a power of two, across various branches. We then analyzed the execution time of core 0 for the different protocols, and show the cache miss rate in different benchmark.

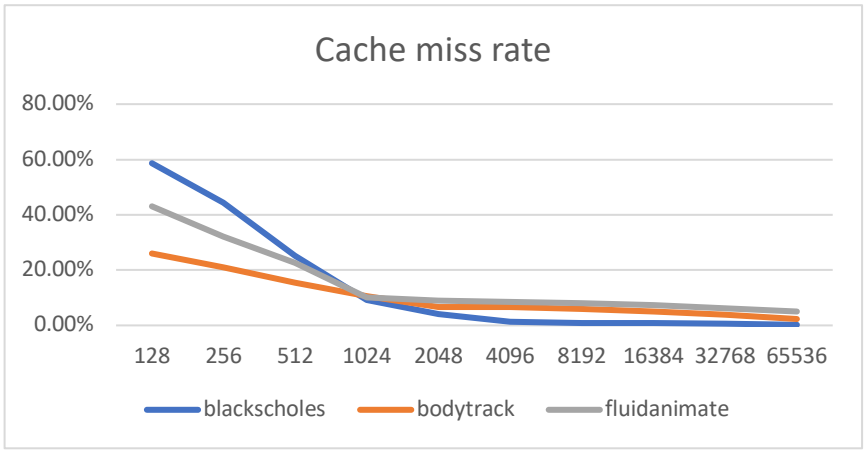
4.1 Cache size

In the experiment, the parameters of 2-way set associativity and a 16-byte block size were held constant. The only variable changed was the cache size during the testing.
The following graphs showed the execution time against cache size in three benchmarks:


Execution time against cache size in blackscholes
2.00E+08
1.50E+08
1.00E+08
5.00E+07
0.00E+00
128
256
512
1024
2048
4096
8192
16384
32768
65536
MESIF
MESI
Dagron


Execution time against cache size in bodytrack
1.20E+08
1.00E+08
8.00E+07
6.00E+07
4.00E+07
2.00E+07
0.00E+00
128
256
512
1024
2048
4096
8192
16384
32768
65536
MESIF
MESI
Dagron

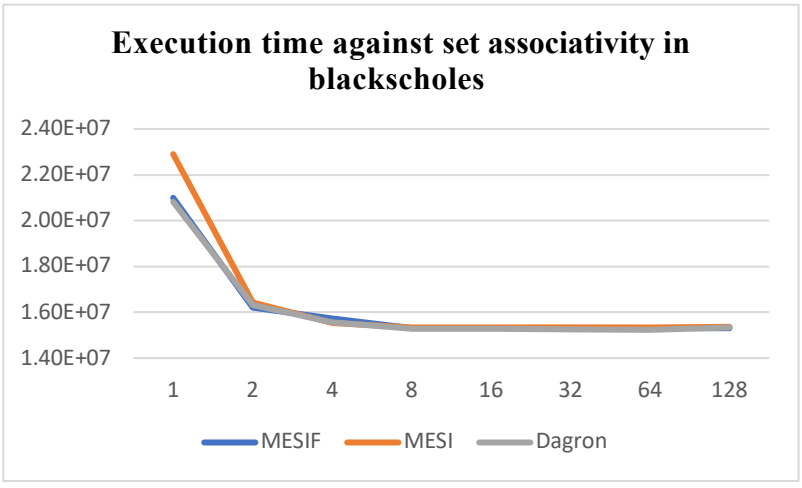

Execution time against cache size in fluidanimate
1.40E+08
1.20E+08
1.00E+08
8.00E+07
6.00E+07
4.00E+07
2.00E+07
0.00E+00
128
256
512
1024
2048
4096
8192
16384
32768
65536
MESIF
MESI
Dagron

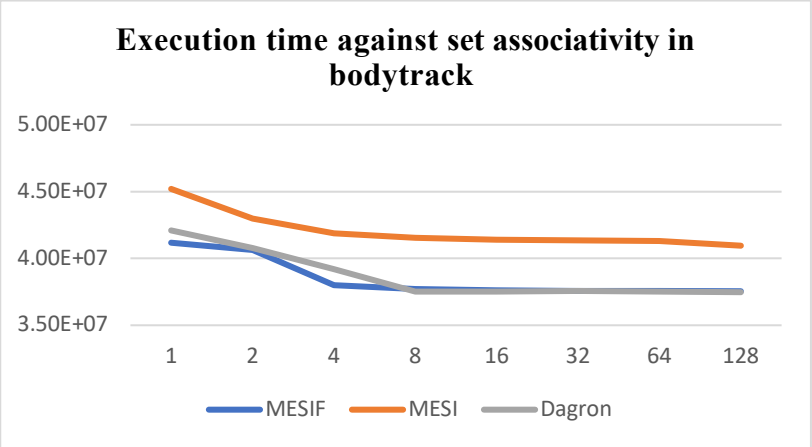


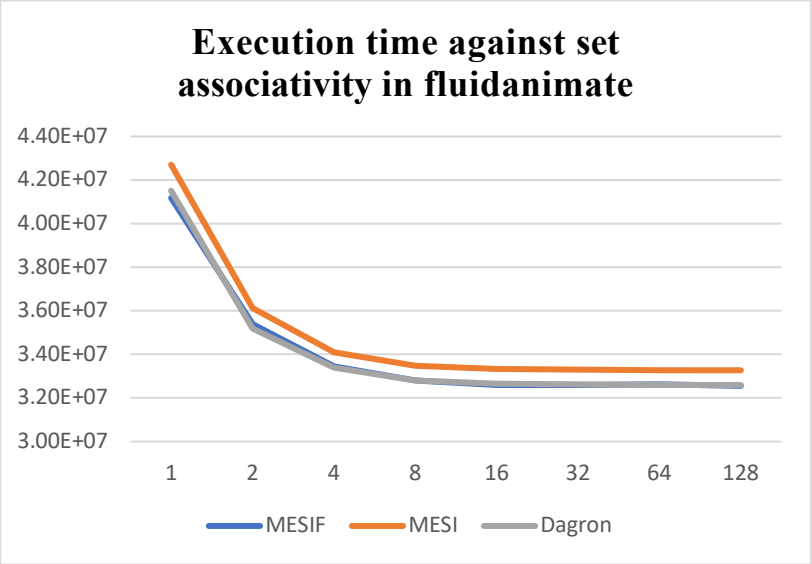
It is obvious the consistent behavior across all cache protocols indicates that increasing the cache size leads to improved performance, characterized by a reduction in the cache miss rate.
However, MESI is the most time-consuming protocol across all benchmarks, given that the classical MESI protocol does not implement cache-to-cache transfers. Since fetching a block from memory takes significantly longer (100 cycles for a 16-byte block) compared to the quick 8-cycle cache-to-cache transfer, the absence of this feature in MESI would naturally lead to longer overall execution times when cache misses occur.

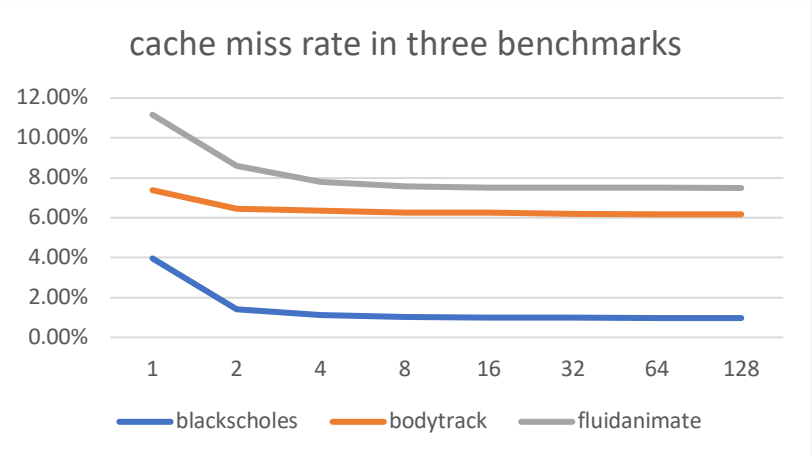
4.2 Set associativity

In the experiment, the parameters of 4096-byte cache size and a 16-byte block size were held constant. The only variable changed was the set associativity during the testing.





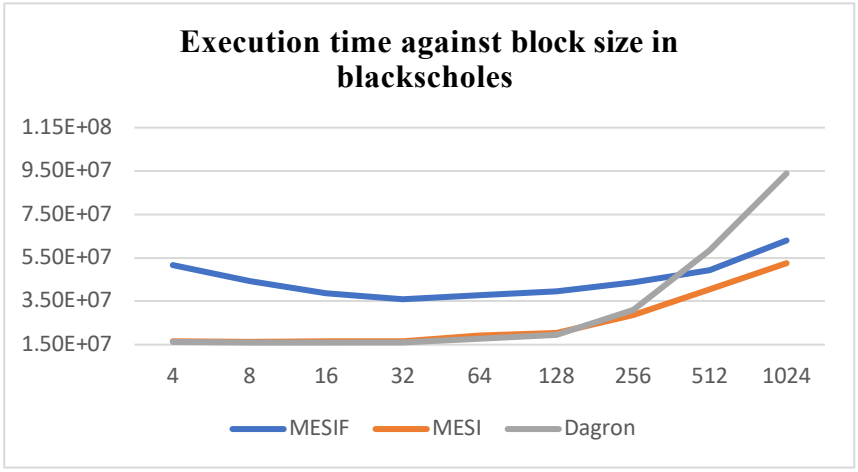


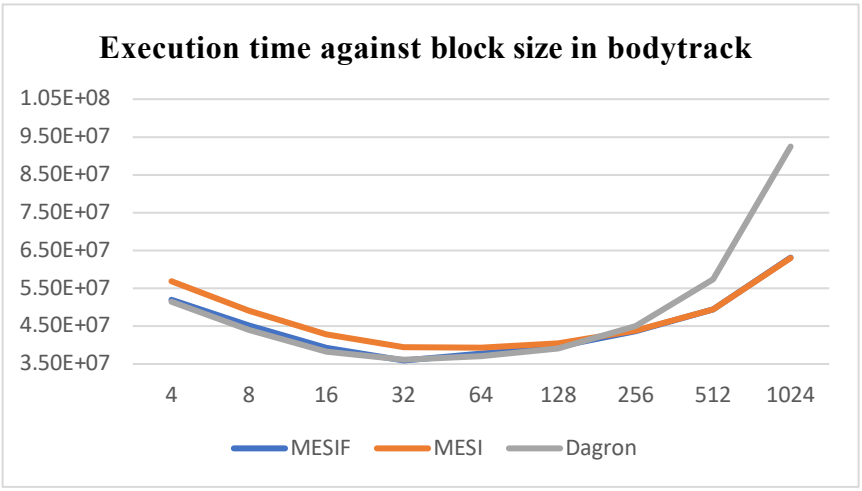


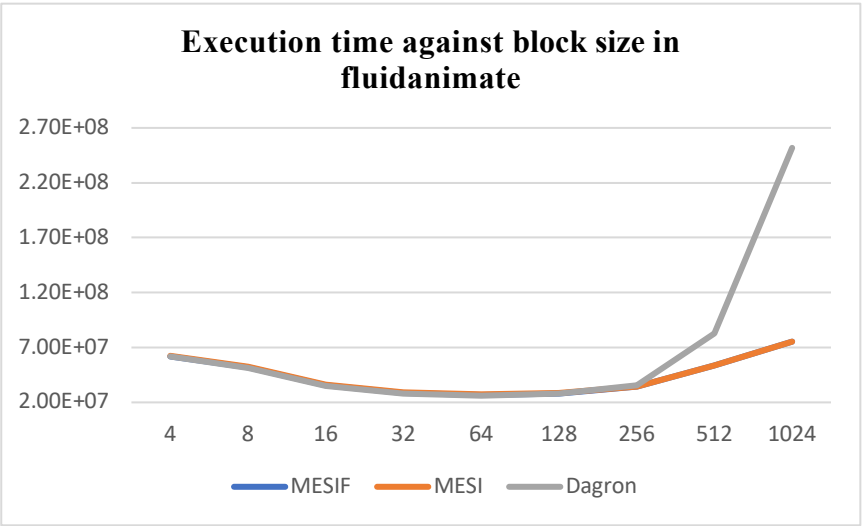
In general, as the value of set associativity increases, performance tends to improve due to a reduction in conflict misses. However, the rate of improvement diminishes as associativity grows, and beyond a certain point, the gains in performance become negligible. Apart from that, the difference in performance between MESI, Dragon and MESIF protocols is the most significant in the bodytrack benchmark.

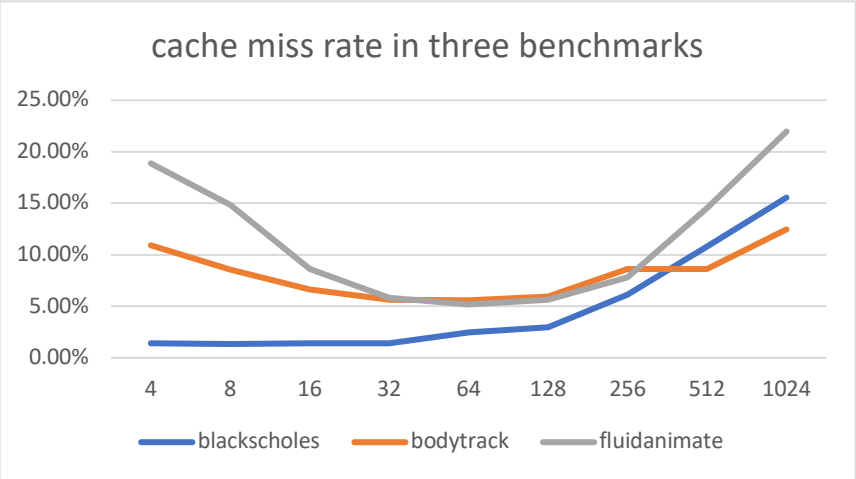
4.3 Block size

In the experiment, the parameters of 4096-byte cache size, 2-way set associativity were held constant. The only variable changed was the block size during the testing.









The test results clearly indicate that there is not a consistent improvement in performance with increasing block size. For the blackscholes benchmark, both the MESI and Dragon protocols experience a rise in execution times with larger block sizes. Although there is a point where larger block sizes do improve performance for the three protocols, this benefit plateaus and eventually reverses, leading to diminished performance.

The cache miss rates follow a similar trend. While initially, a larger block size can decrease the miss rate, there is a critical point beyond which any further increase does not yield additional benefits and may, in fact, degrade performance. This suggests that there is an optimal block size for each protocol and benchmark, and exceeding this size can be counterproductive.

5 Conclusion

The experiment helped our grasp of cache coherence concepts and enabled us to simulate and assess the performance of different cache coherence protocols. Through this practical exploration, we gained insights into how these protocols manage data consistency across multiple cache instances in a computing system.

Experimental results showed that larger cache sizes improve the performance, but associativity only improves performance to a limited extent. Moreover, the performance enhancement observed between the Dragon and MESIF protocols was marginal, suggesting that both protocols perform comparably under the tested conditions.