CS4223 Multi-Core Architectures
AY20/21 Sem 1

Cache Coherence Simulator for MESI, Dragon and MESIF Protocols

15 November 2020

Nguyen Dang Phuc Nhat (A0184583U)
Tran Tan Phat (A0170740M)
Keven Loo Yuquan (A018338Y)

Table of Contents

1 Implementation Language

Our group decided to implement the simulator with an OOP design, and thus C++ and Java were the two most appropriate implementation languages.

However, a Java implementation would not be able to match its equivalent C++ counterpart in real time execution speed. Thus, we selected C++ to implement our cache coherence simulator.

2 Simulator Design and Implementation

2.1 Simulator

Our simulator employs an OOP design, with each component in the system modelled by its own class. The program driver is **`main.cpp`**, which accepts 5 command-line inputs - the protocol, benchmark, cache size, associativity and block size, and parses them accordingly.

The simulator requires that:

- The specified cache coherence protocol is one of: MESI, Dragon or MESIF
- Cache and block sizes are multiples of the default word size (for word-aligned accesses)
- Caches are divided into an integer number of sets by the specified associativity
- The simulator is executed from the same directory or a sub-directory

If the input satisfies the above, the correct **`Runner`** will be invoked to perform the simulation.

The **`Runner`** for each protocol encodes all the states, transitions and bus transactions of that protocol. The same implementation of the processors, caches and bus is used for all three protocols.

2.2 Base devices

A hardware component connected to the shared bus is represented as a **`Device`**. An object of **`Device`** class only has 2 states: `Free` and `Busy`, and each object keeps track of `nextFree`, the time it is next `Free`.

When a **`Device`** is `Free`, it is allowed to initiate a new transaction on the shared bus. A **`Device`** in the `Busy` state becomes `Free` when it completes its current action (e.g computation), or is no longer waiting on a bus transaction.

When a **`Device`** is `Busy`, it is not allowed to initiate a new transaction on the shared bus. However, it still snoops on the shared bus and is able to respond to bus activity when necessary. A **`Device`** enters the `Busy` state if it performs computation or initiates a new bus transaction.

2.3 Processor cores and caches

Processor cores are modelled by the **`Core`** class, which inherits from **`Device`**. Thus, a **`Core`** possesses the same states as those discussed in the **`Device`** section. In addition, a **`Core`** stores its own execution statistics and its progress on its instructions.

The L1 data caches are modelled by the **`Cache`** class. A **`Cache`** is stateless, and comprises many instances of **`CacheEntry`**. Each **`CacheEntry`** represents a cache block and has a `state` that is managed by the **`Runner`**.

- Cache type: n-way set-associative cache
    - A direct-mapped cache can be emulated by specifying associativity of 1
    - A fully-associative cache can be emulated with the maximum associativity
- Replacement policy: Least Recently Used (LRU) - which is supported by storing an additional `lastUsed` value for each **`CacheEntry`**.

Each **`Core`** is paired with a **`Cache`**. This connection is not explicitly reflected in these classes, but handled by the **`Runner`**. The memory controller is our simulator is handled implicitly by the **`Runner`**, and serves as the backing source for clean data blocks.

Our implementation of the L1 data caches takes the cache size to refer to the amount of data stored in the cache, and thus excludes any bits required to store additional state for each block.

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. In addition, we assume the memory controller employs a write-back buffer to hold invalidated or evicted blocks requiring a write-back to memory.

2.4 Bus

The bus in our simulator is modelled by the **`Bus`** class, which stores its own statistics on bus traffic and the composition of bus transactions.

We considered implementing our bus as an *atomic bus*, where only one cache communicates at a time. The bus is locked for the full duration of a bus transaction, but other components are allowed to respond to the transaction. However, whilst simple to implement, this incurs a heavy performance penalty as the bus will idle whilst the response to the transaction is pending. This decreases effective bandwidth, and artificially serialises memory requests by different caches to different data blocks.

Therefore, we implemented our bus as a **split transaction bus**, with each bus transaction split into two smaller transactions - a request and a response. Due to the latency between request and response, other transactions are allowed to intervene between a particular request and response. This enables *pipelining* of bus transactions and *out-of-order completion* of requests, improving bandwidth. Thus, the request order establishes the total order for the system. We assume the request bus allows one request to be placed per cycle, and that the response bus (for data) has a sufficiently large bandwidth.

Each cycle, the bus arbiter (modelled in the **Runner**) allows only one cache to place a new request on the bus. To **minimise starvation**, we employ a priority-based heuristic that selects the cache whose last bus request has the oldest time, in effect penalising frequent bus users. During this cycle, a request table entry is allocated for the new request, the snooping caches update their cache state. If necessary, the corresponding cache/memory designated to supply/receive a data block will also start preparing the response.

To prevent caches from making conflicting requests to a block involved in a pending request, we track outstanding requests (up to four) in a *request table* in **Runner** (the bus arbiter). A cache that attempts to place a request for a pending block will be stalled until the response for that pending block completes.

In our design, bus clients are not blocked on requesting access to the bus, and are able to service incoming transactions (e.g. cache-to-cache transfers) whilst waiting to issue requests. This is to prevent *fetch-deadlock*, e.g. a cache blocks with an exclusive read-write copy of a line, locking out other caches from accessing this line

Consider the following scenario in the MESI protocol where processors A and B are writing to the same cache line B: P1 issues a BusRdX; P2 invalidates; Before P1 completes its cache line update, P2 issues its BusRdX; P1 invalidates, and this repeats, resulting in a *livelock*. To prevent livelock in such cases of heavy contention, our bus grants a cache requesting a block to *complete* one operation before another request for this block is granted by the bus arbiter, guaranteeing *progress*.

The memory controller in our system employs a write-back buffer to store pending writes of invalidated or evicted blocks to memory, but does not examine the contents of the buffer against incoming requests. Therefore, if cache-to-cache transfer is not available (e.g. classic MESI protocol), or the last copy of a block has been evicted, a new request for that block will have to wait for that block to be written back fully before the memory transaction (taking 100 cycles) can begin.

2.5 Runners

A **Runner** is responsible for simulating cache coherence protocols. It is aware of all components in the system: 1 common **Bus**, 4 **Core** instances, 4 **Cache** instances.

As mentioned, to prevent conflicting requests, the **Runner** keeps track of cache blocks that are involved in transactions (both memory-cache and cache-cache transactions), and stalls a **Core**'s write hit and read/write misses if the requested address belongs to a block involved in a pending request.

There are 3 subtypes of **Runner** implemented:

- **MESIRunner**: Simulates the classic no-intervention MESI protocol.
- **DragonRunner**: Simulates the Dragon protocol.
- **MESIFRunner**: Simulates the MESIF protocol.

Each subtype Runner implements the following scenarios:

- `simulateReadHit`
- `simulateReadMiss`
- `simulateWriteHit`
- `simulateWriteMiss`

The simulation is carried out cycle by cycle.

At each cycle, the **`Core`**s are sorted: only **`Core`**s in `Free` state are processed. To avoid starvation of bus access, the **`Runner`** sorts the **`Core`** based on the last time they obtained access to the **`Bus`**. The more recent a **`Core`** has had access to **`Bus`**, the lower its priority is. If two **`Core`**s are equal in this metric, tie-breaking is based on their respective ID - the **`Core`** with the smaller ID will get the priority.

After sorting, the **`Runner`** will process each **`Core`**'s next trace (if execution is possible). A **`Core`**'s trace might be stalled due to several 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 the availability time is `INF`, the block's latest copy is guaranteed to be stored in some **`Cache`** of some **`Core`**.

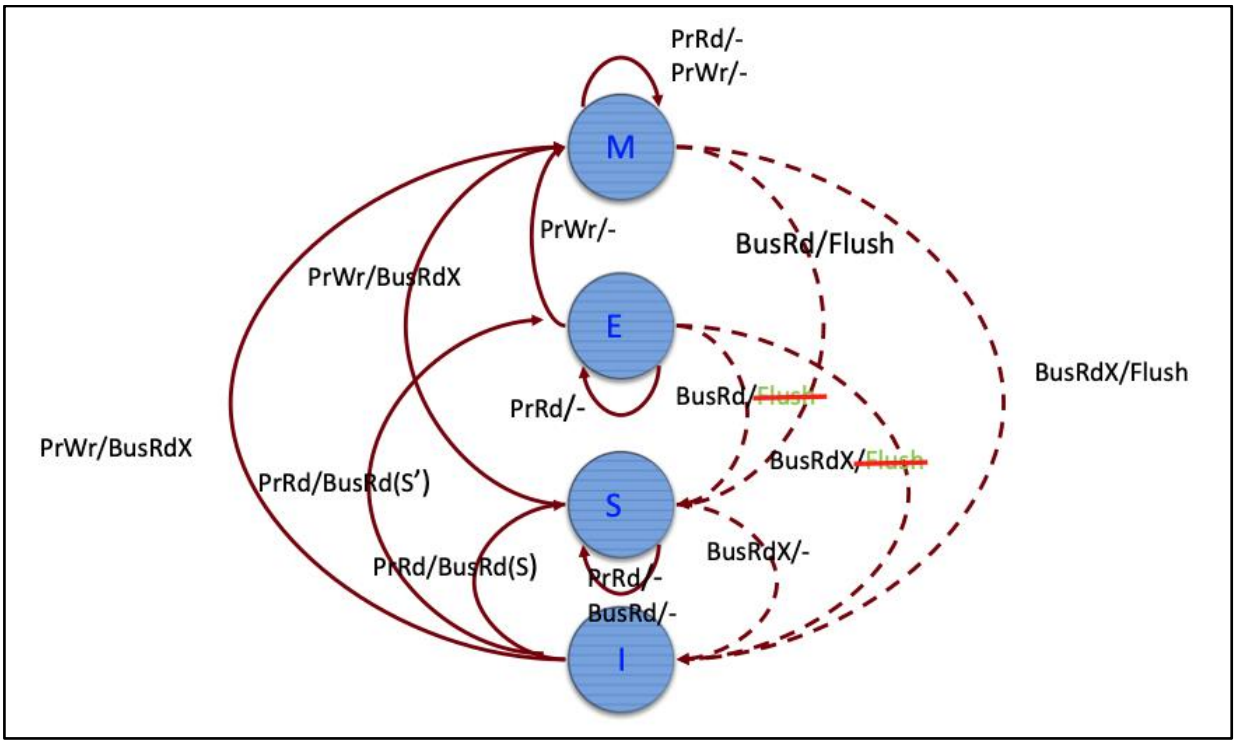
3 Cache Coherence Protocols

3.1 MESI invalidation-based protocol

This protocol is implemented in the **`MESIRunner`** class, with four states: Modified (M), Exclusive (E), Shared (S) and Invalid (I). The variant implemented is classical MESI with no intervention, i.e. no cache-to-cache transfers from M or E state.

Thus, all misses for a block are serviced directly by the memory. If another cache previously held the block in M state, the additional latency from the write-back triggered is accounted for.

The state transition diagram for the MESI protocol is shown below, along with the table of permitted states for a pair of caches in the MESI protocol.



MESI	M	E	S	I
M	No	No	No	Yes
E	No	No	No	Yes
S	No	No	Yes	Yes
I	Yes	Yes	Yes	Yes

To simulate a given processor's instruction in a given cycle:

- Type 0: Load
  - `simulateReadHit:` The **`Core`** is set to `Busy` for 1 cycle.
  - `simulateReadMiss:` If the requested block is not in transaction, and the **`Bus`** is available, proceed to issue a `BusRd` operation. If the block is in state `M` in some other **`Cache`**, there will be a stall until the block has been flushed to the memory.

The block is then fetched from memory. The **Core** is set Busy until the block has been filled in the **Cache**.

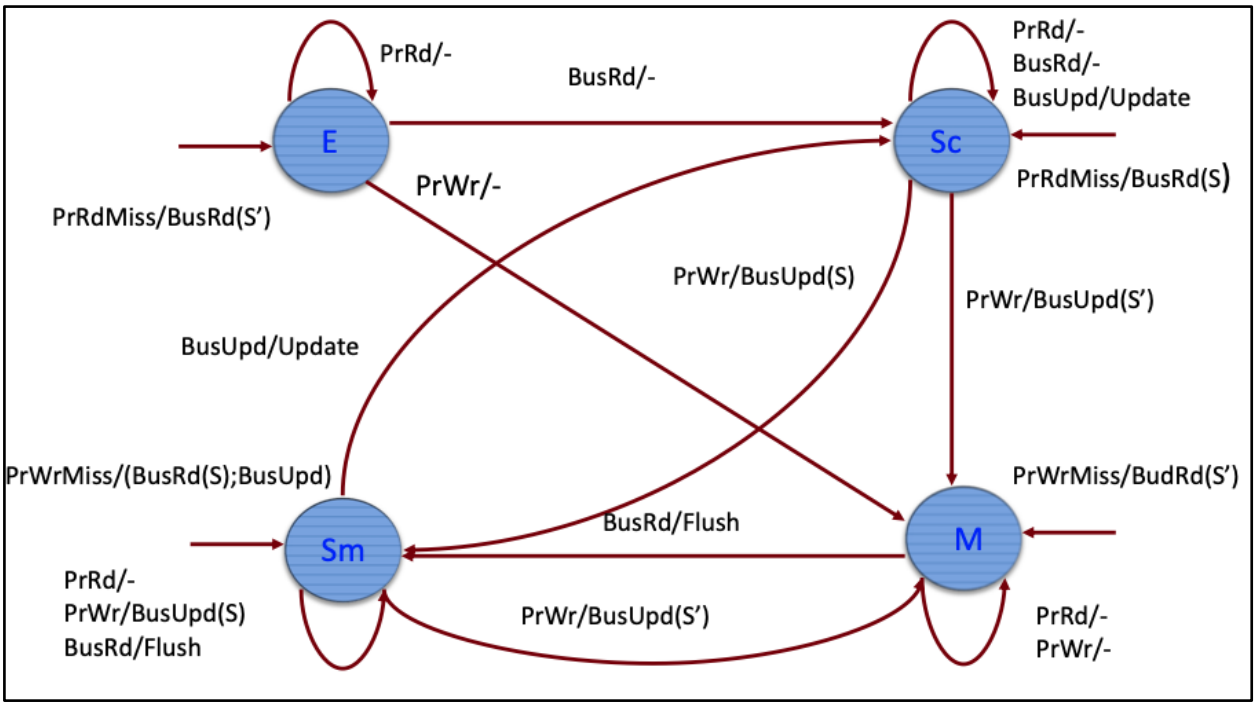
- Type 1: Store
  - simulateWriteHit: The **Core** is set to Busy for 1 cycle.
  - simulateWriteMiss: If the requested block is not in transaction, and the **Bus** is available, proceed to issue a BusRdX operation. If the block is in state M in some other **Cache**, there will be a stall until the block has been flushed to the memory. As a result of BusRdX, clean copies of the block stored in other **Cache**s would be invalidated. The block is fetched from memory. The **Core** is set Busy until the block has been filled in the **Cache**.
- Type 2: Compute
  - Simply set the **Core** to Busy for the required number of cycles.

3.2 Dragon update-based protocol

The protocol is implemented in the **DragonRunner** class, with four states: Modified (M), Exclusive (E), Shared Modified (Sm) and Shared Clean (Sc). As per the protocol, cache-to-cache transfer is implemented.

As mentioned previously, if an evicted block needs to be written back to memory, we assume it can be done concurrently with data fetching with the combined use of a write-back buffer and line-fill buffer.

The state transition diagram for the Dragon protocol is shown below, along with the table of permitted states for a pair of caches in the Dragon protocol (next page).



Dragon	E	Sc	Sm	M
E	No	No	No	No
Sc	No	Yes	Yes	No
Sm	No	Yes	No	No
M	No	No	No	No

To simulate a given processor's instruction in a given cycle:

- Type 0: Load
  - `simulateReadHit:` The **`Core`** is set to `Busy` for 1 cycle.
  - `simulateReadMiss:` If the requested block is not in transaction, and the **`Bus`** is available, proceed to fetch and allocate a copy of the block from: memory if no other caches have a copy, and from another cache otherwise. The **`Core`** is set Busy until the block has been filled in the **`Cache`**.
- Type 1: Store
  - `simulateWriteHit:` The **`Core`** is set to `Busy` for 1 cycle. If the block is not in state M, there would be broadcasting of the written word - which takes 2 cycles.
  - `simulateWriteMiss:` If the requested block is not in transaction, and the **`Bus`** is available, proceed to fetch and allocate a copy of the block from: memory if no other caches have a copy, and from another cache otherwise. The **`Core`** is set Busy until the block has been filled in the **`Cache`**. Finally, the written word might be broadcasted, taking 2 cycles. In those 2 cycles, the block is considered in a transaction.
- Type 2: Compute
  - Simply set the **`Core`** to `Busy` for the required number of cycles.

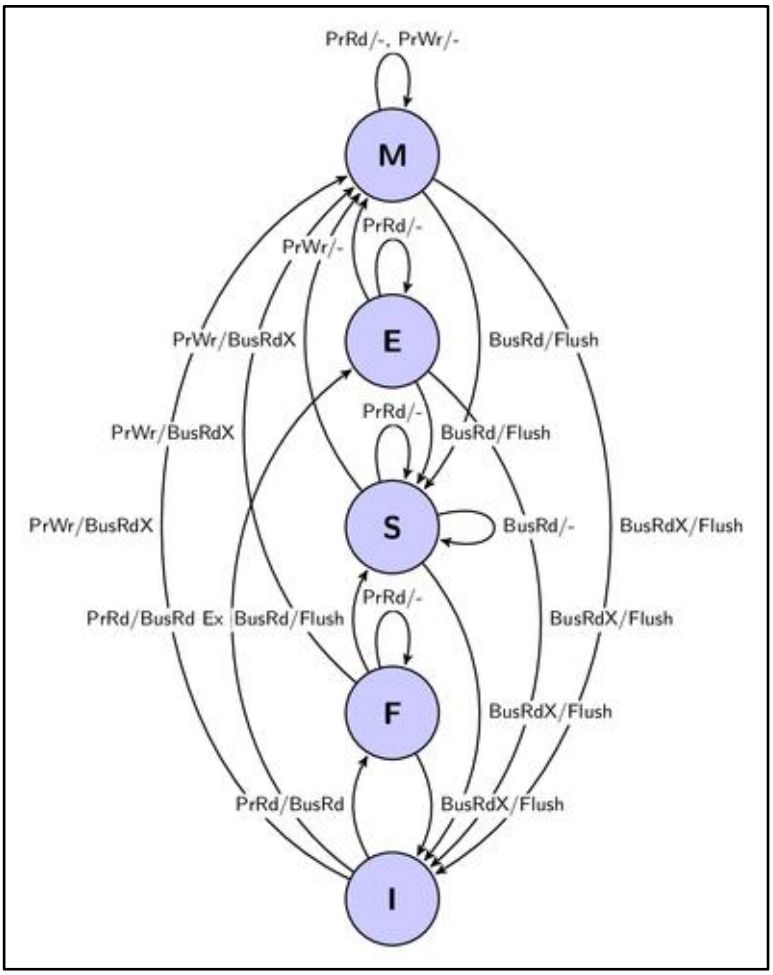
3.3 Advanced task: MESIF invalidation-based protocol

In the MESI protocol, a request for a cache block received by multiple caches holding the block in the S state will be serviced inefficiently - it may be satisfied from main memory (classical MESI) or all the caches could respond, overloading the data bus (Illinois MESI).

MESIF is an opportunistic optimisation of the Illinois variant of MESI in commercial use by Intel for cache-coherent NUMA architectures. It introduces the F state as a specialised form of the S state. The F state designates the cache as the responder for any requests for that block, eliminating redundant responses from other caches and saving bandwidth on the bus. All caches in the S state are now silent, and the cache whose request is served is assigned the F state (if it is not the only copy in the system - where it transitions to E or M instead).

This protocol is implemented in the **`MESIFRunner`** class, with *five* states: Modified (M), Exclusive (E), Shared (S), Invalid (I) and Forward (F). The variant implemented extends classical MESI with intervention (cache-to-cache transfers from M or E states in a `FlushOpt` transaction) to include the F state.

The state transition diagram for the MESIF protocol is shown below, along with the table of permitted states for a pair of caches in the MESIF protocol.



MESIF	M	E	S	I	F
M	No	No	No	Yes	No
E	No	No	No	Yes	No
S	No	No	Yes	Yes	Yes
I	Yes	Yes	Yes	Yes	Yes
F	No	No	Yes	Yes	No

The following changes were made to the classical no-intervention MESI implementation to simulate a given processor's instruction:

- Type 0: Load
  - `simulateReadMiss:` The block is fetched from: memory if no other caches have a copy (after all write-backs for that block complete), or otherwise the faster of memory and the forwarding cache in M, E or F states (if any exist); this cache is then designated the F state if other shared copies exist, or otherwise the E state
- Type 1: Store
  - `simulateWriteMiss:` The block is fetched from: memory if no other caches have a copy, or otherwise the faster of memory and the forwarding cache in M, E or F states (if any exist); this cache is then designated the M state since it acquired the block with intent to write

4 Workload Analysis

For the simulator, the execution traces for the benchmarks **`blackscholes`**, **`bodytrack`** and **`fluidanimate`** from the PARSEC 2.0 benchmark suite were provided.

This benchmark suite was primarily designed to assess the performance of chip multi-processors (CMPs) with a set of multi-threaded workloads. As the workloads are diverse and span different application domains, it is useful to identify their execution characteristics as they may inform us of their performance with different cache coherence protocols.

The table below summarises the characteristics of the three benchmarks, as reported by the PARSEC 2.0 documentation.

Benchmark	Parallelisation		Working Set	Data Usage	
	Model	Granularity		Sharing	Exchange
`blackscholes`	Data-parallel	Coarse	Small	Low	Low
`bodytrack`	Pipelined	Medium	Medium	High	Medium
`fluidanimate`	Data-parallel	Fine	Large	Low	Medium

In particular, the workloads are implemented in the following way:

- **`blackscholes`**: A master thread initialises the portfolio data and then spawns worker threads that each process an independent section of the data - the working set is small and there is minimal communication between threads except with the master.
    - This benchmark is expected to perform similarly regardless of protocol, since there is minimal data sharing (implying most misses are served by memory) and little contention for shared data that would result in invalidations or updates.
- **`bodytrack`**: Multiple processes cooperate to process the same particle filter data in a pipelined fashion for increased concurrency.
    - This benchmark is expected to perform better on protocols with cache-to-cache transfers due to the large extent of data sharing and exchange; additionally, it is likely to perform better with larger caches due to reduced capacity misses from the moderate working set.
- **`Fluidanimate`**: Multiple threads cooperate to approximate a continuous fluid with a list of discrete particles; each thread takes a subset of particles and communicates with threads containing neighbouring particles.
    - This benchmark is expected to perform better on update-based protocols and protocols with cache-to-cache transfers due to the extent of data exchange between fluid simulation cycles; additionally, it is likely to perform better with larger caches due to reduced capacity misses from the large working set.

5 Experimental Results and Analysis

All experiments have been run by fixing 2 of the three parameters (cache size, associativity, block size) to their default values and varying the other. The default parameters are: 4096-byte cache size, 2-way set associativity and a 16-byte block size.

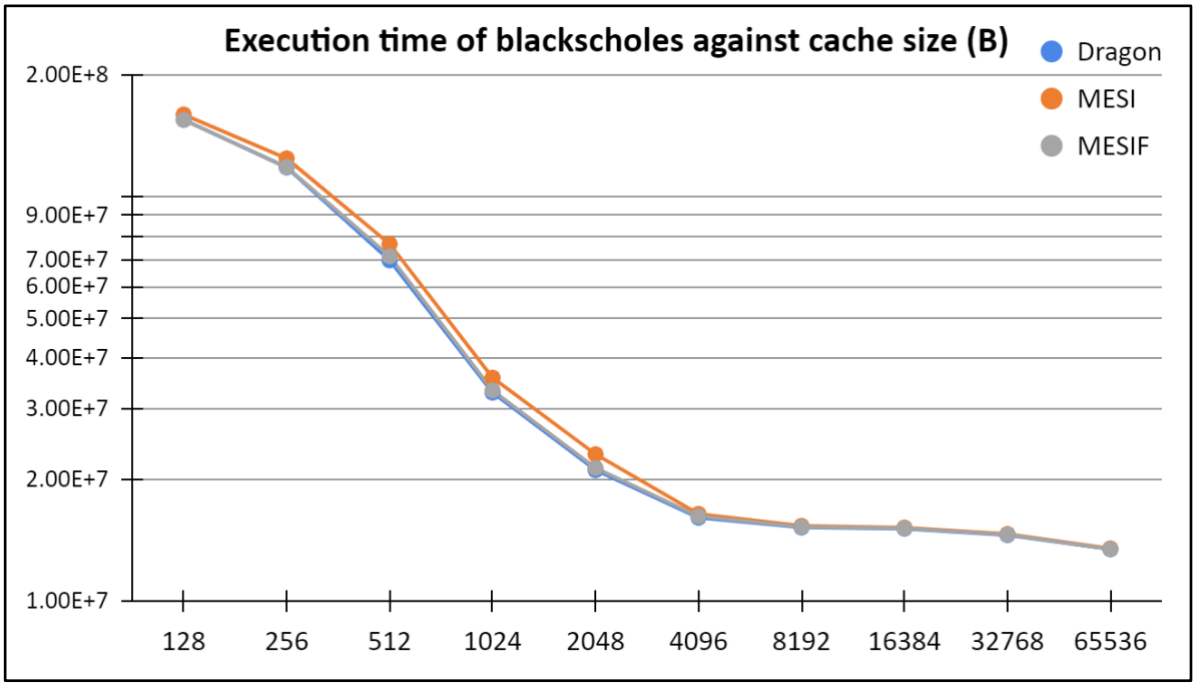
5.1 Cache size

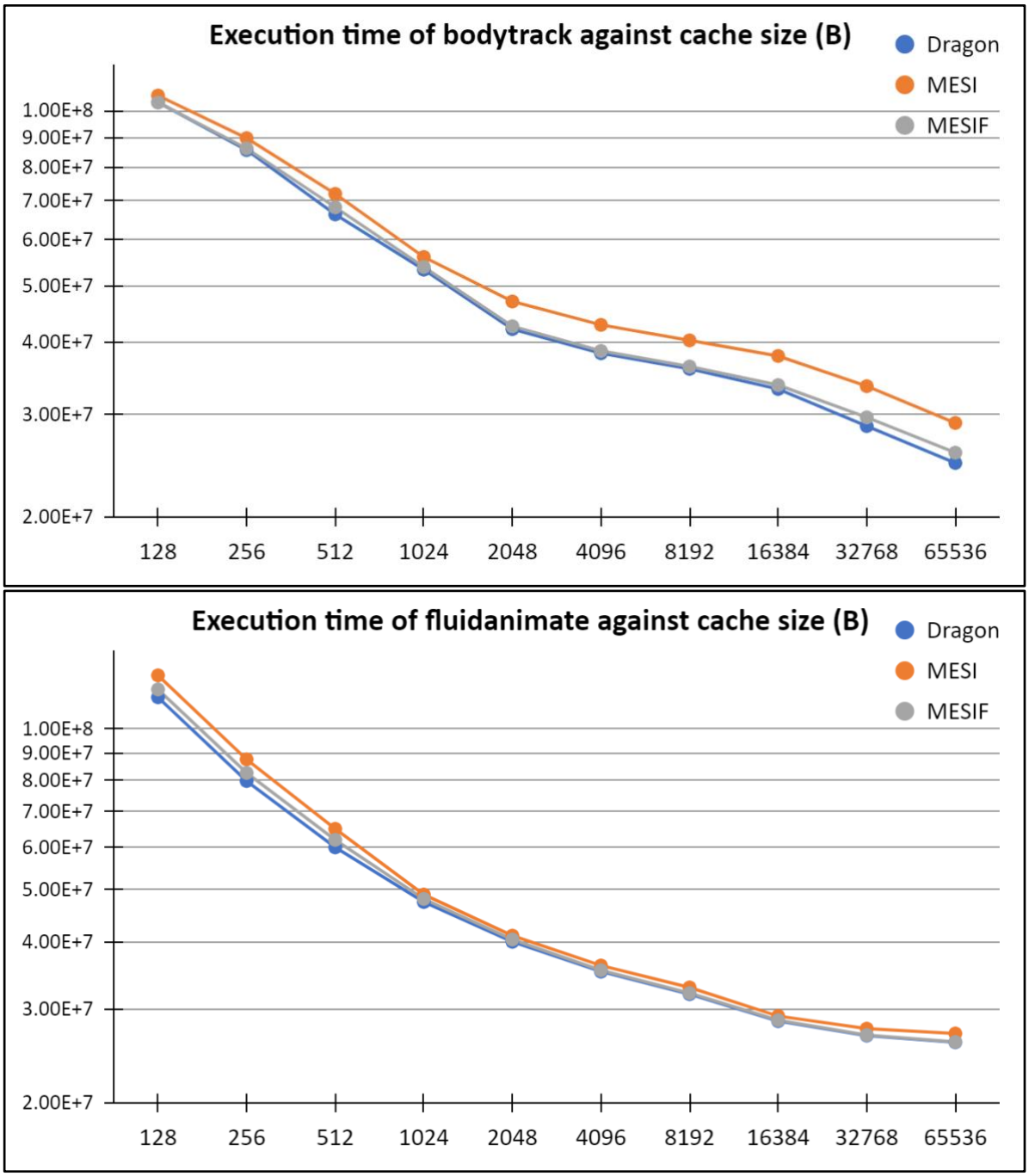
In these runs, cache size is the varying parameter.

The general trend for all 3 protocols is that larger cache size can help reduce runtime, by reducing the

For all benchmarks, MESI has always been the most time-consuming protocol. This is an expected result as cache-to-cache transfers were not implemented in the classic MESI protocol. As block size is kept at 16 bytes in these runs (equivalent to 4 words per block), the time to fetch a block from memory is 100 cycles, significantly longer than the 8-cycle cost of transferring a copy from an existing cache.

Dragon and MESIF have had similar performance, with Dragon edging over MESIF in some instances. The results broadly verify our hypotheses from Section 4.



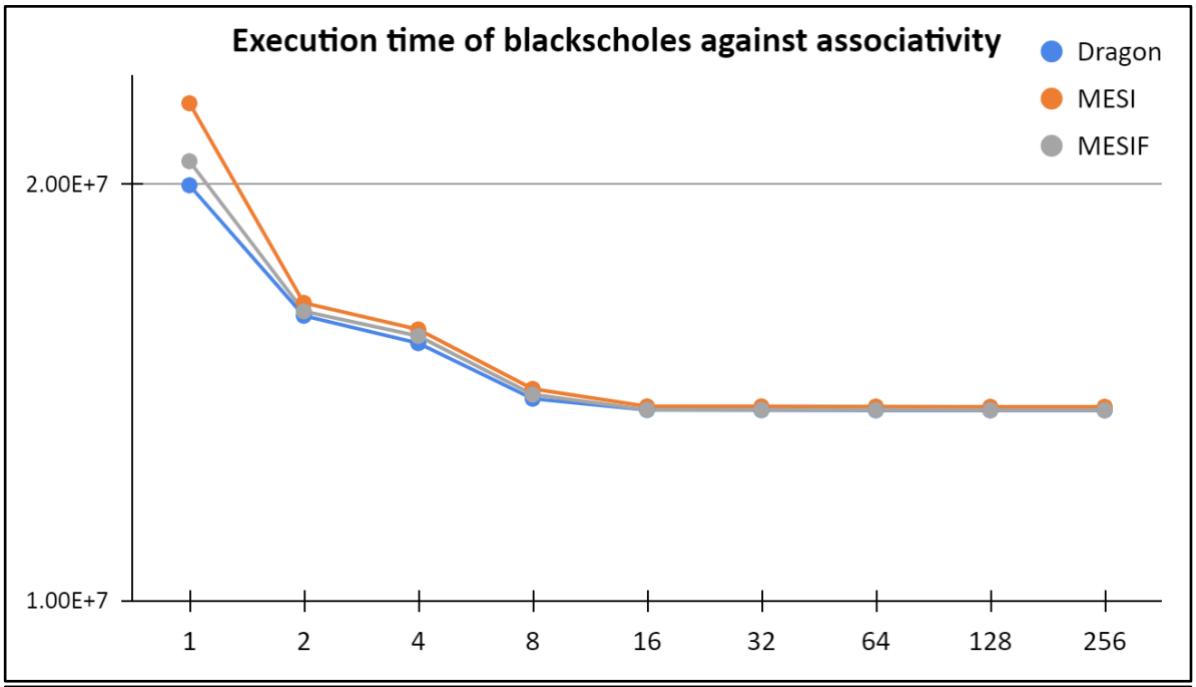


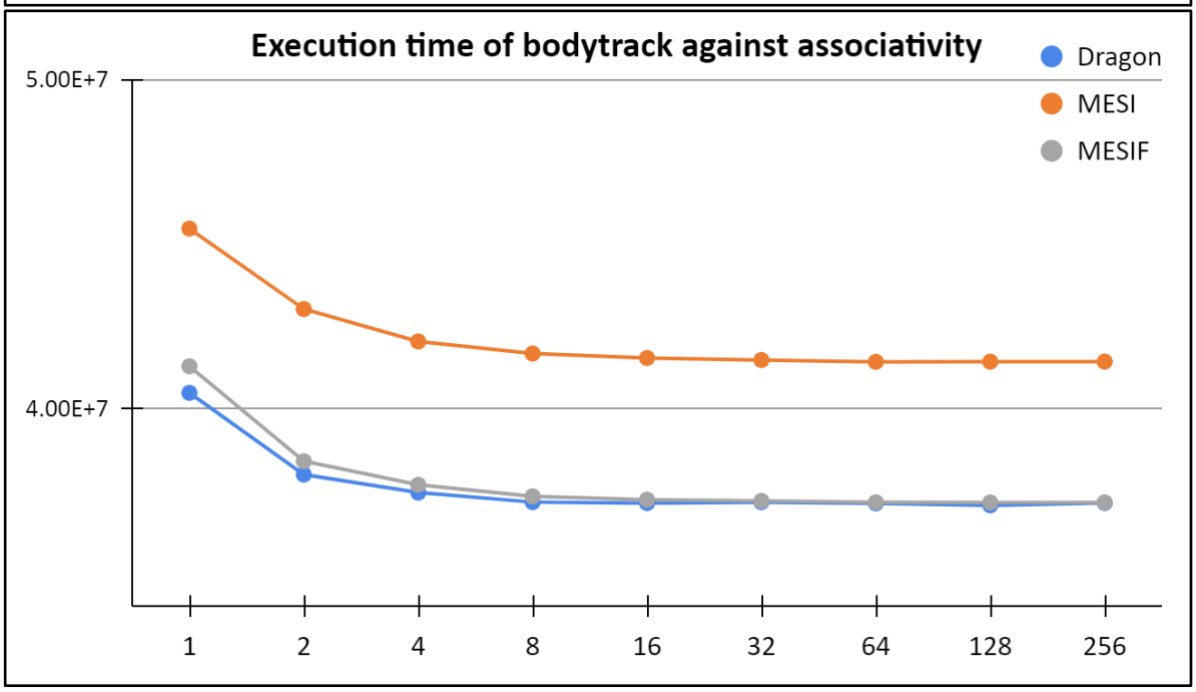
5.2 Set associativity

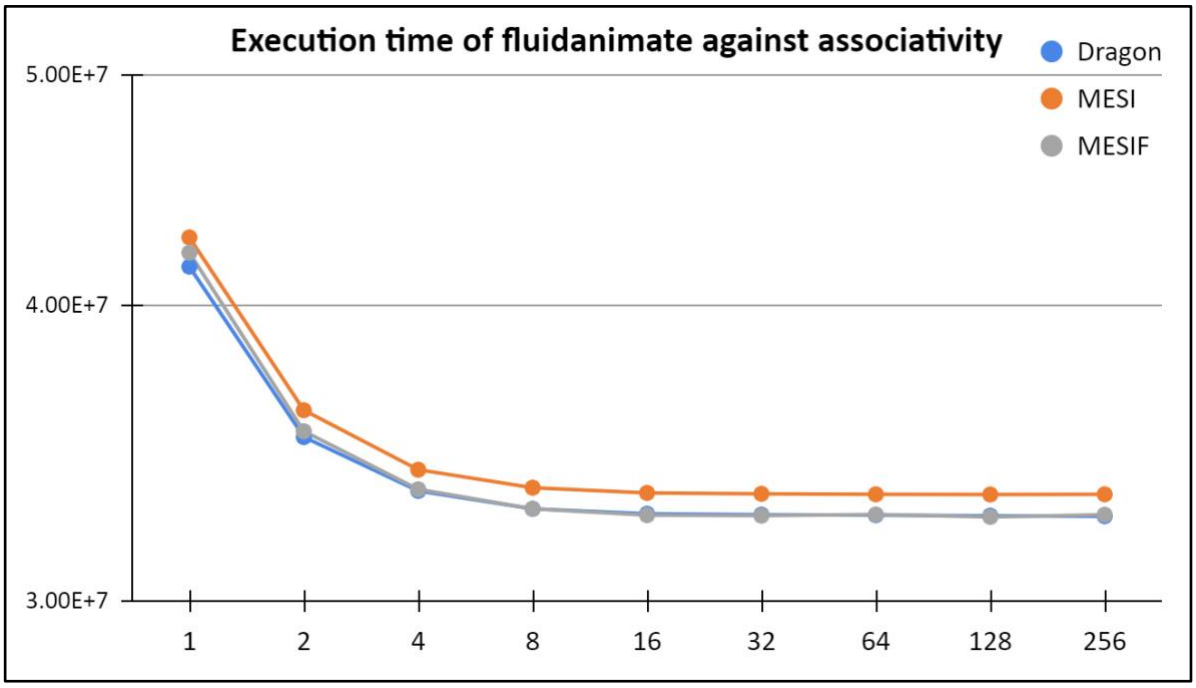
In these runs, set associativity is the varying parameter.

The general trend is again increasing the parameter leads to better performance. However, improvement rate decreases substantially as associativity gets larger (due to reduced conflict misses) and eventually, performance gain is unnoticeable.

The inefficiency of MESI protocol is again highlighted in these experiments. The disparity between MESI and Dragon/MESIF is most significant for the **`bodytrack`** benchmark.


Execution time of blackscholes against associativity
Dragon
MESI
MESIF
2.00E+7
1.00E+7
1
2
4
8
16
32
64
128
256


Execution time of bodytrack against associativity
Dragon
MESI
MESIF
5.00E+7
4.00E+7
1
2
4
8
16
32
64
128
256

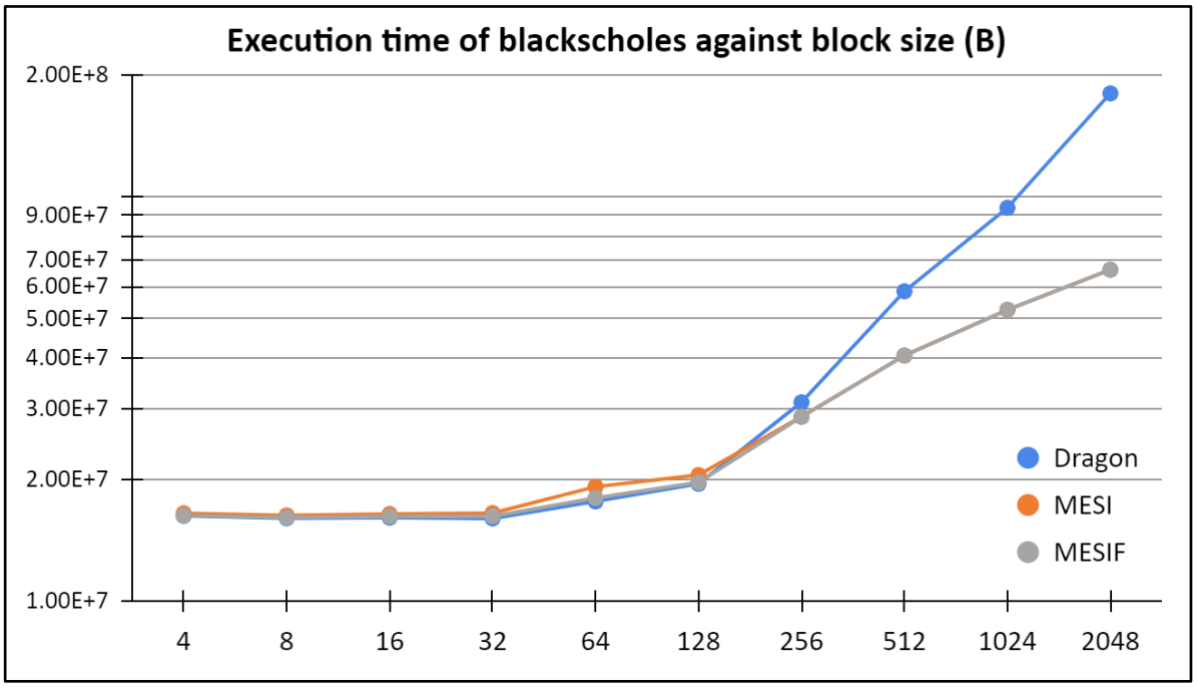


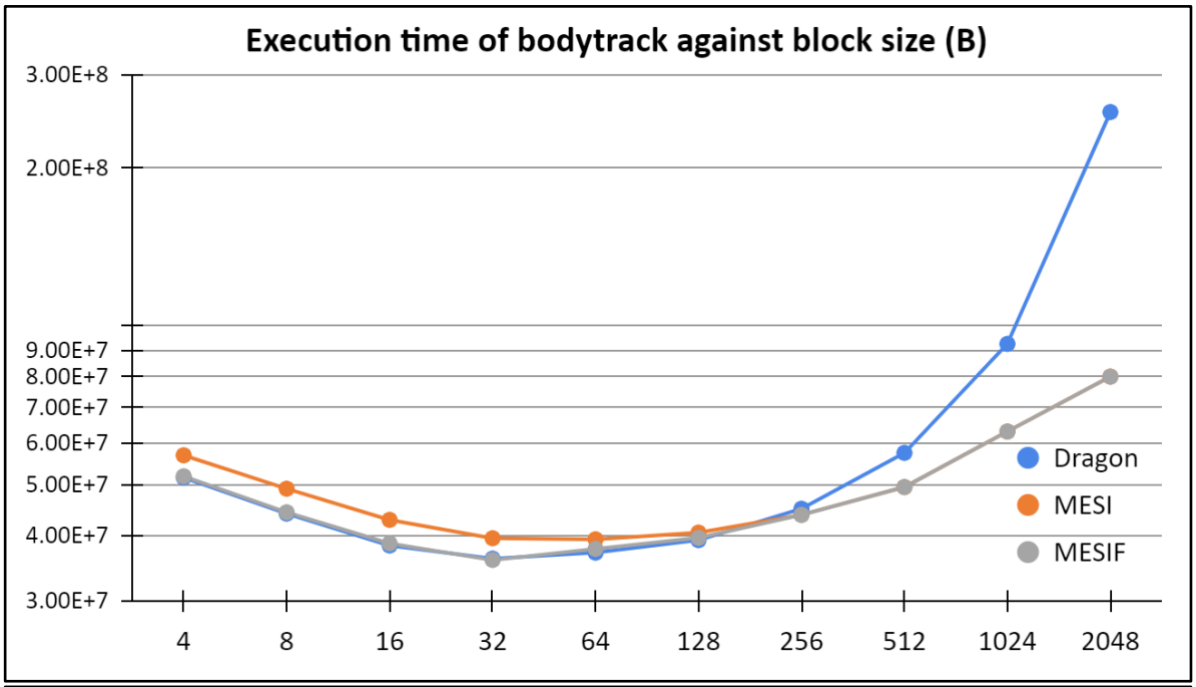
5.3 Block size

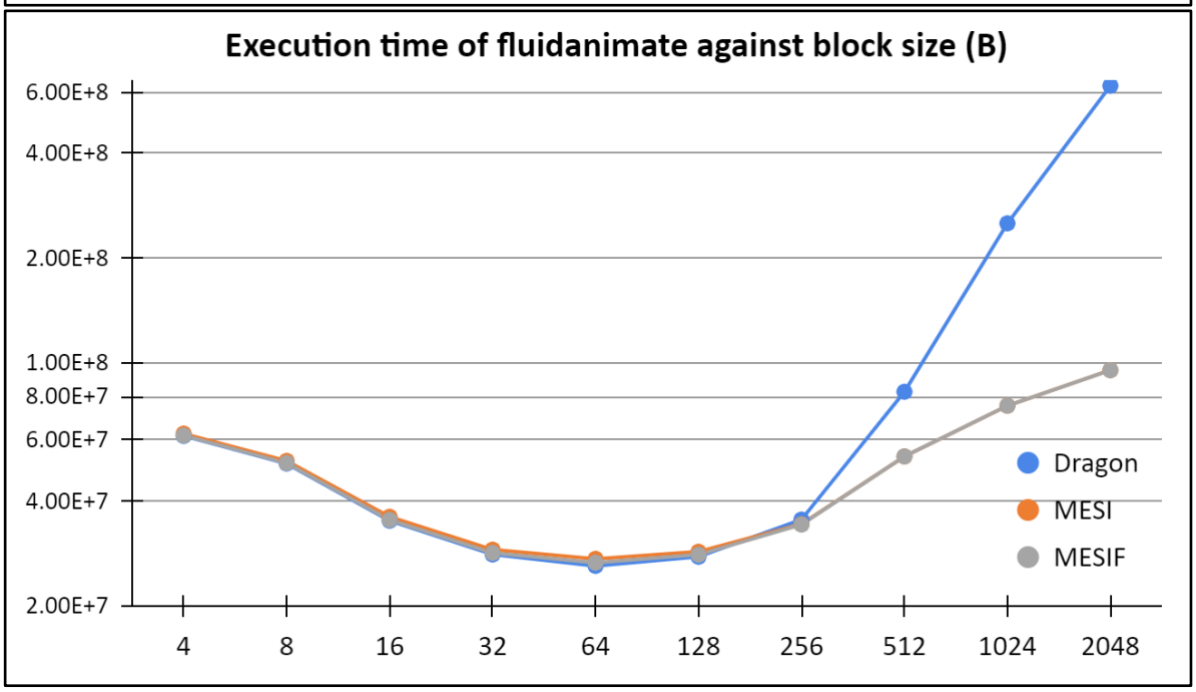
In these runs, block size is the varying parameter.

Having larger block size only benefits in two benchmarks **`bodytrack`** and **`fluidanimate`**. For the **`blackscholes`** benchmark, larger block size hinders runtime. However, any gain from larger block size could only go so far, before placing a huge burden on performance.

For Dragon, performance gets worse at an incredibly high rate and ends up being the worst protocol in latter cases. It is most likely because larger block size results in significant cache-to-cache block transfer time, and this time can exceed the static time of 100 cycles to fetch a clean block from memory.




Execution time of bodytrack against block size (B)
3.00E+8
2.00E+8
9.00E+7
8.00E+7
7.00E+7
6.00E+7
5.00E+7
4.00E+7
3.00E+7
4
8
16
32
64
128
256
512
1024
2048
Dragon
MESI
MESIF


Execution time of fluidanimate against block size (B)
6.00E+8
4.00E+8
2.00E+8
1.00E+8
8.00E+7
6.00E+7
4.00E+7
2.00E+7
4
8
16
32
64
128
256
512
1024
2048
Dragon
MESI
MESIF

6 Conclusion

This project has allowed us to explore various problems that come with implementing simulators. Many decisions and assumptions have to be made as a result. Stalling a component is seen as a simple way to ensure correctness and ease of implementation.

From the experiments, it has confirmed the benefit of having larger cache size, though it does not suggest larger cache is strictly equal to better performance. Set-associativity and block size should only be increased with moderation. It has also been shown that cache-to-cache transaction is a great implementation choice for a cache coherence protocol. As far as performance is concerned, the difference between Dragon and MESIF has been insignificant.