CS4223 Project Report

Introduction

We implemented a cache simulator to analyse how different snooping-based coherence protocols such as MESI, MOESI, MESIF, and Dragon, perform under various workloads. Given any program, we can use our simulator to compare the performance of various protocols, based on the number of Bus Transactions, Memory Requests, Memory Write-Backs and Cache-to-Cache Transfers.

Approach

Simulated Hardware Architecture

We use a system-wide bus to broadcast bus transactions to all caches and main memory. The bus is **atomic** i.e. only one bus transaction can be in flight at any given time. This is a simplification of the actual bus architecture which is typically done as a *split-transaction bus*. In our implementation, therefore, the bus is essentially *locked* until the transaction is completed.

Caches use the Least-Recently-Used (LRU) eviction policy and are *write-allocate write-back*. All processors are assumed to have the same cache configuration i.e. same size and associativity. Any cache-to-cache transfer of a word of data takes 2 cycles, and any cache-to-cache transfer of a cache line takes 2N cycles, where N is the number of words in a cache line. Addresses are 32 bits and there are 4 processors by default.

Writes and reads to the main memory are assumed to take 100 cycles each. There is no write buffer for writes to the main memory. Bus arbitration is done with a FIFO queue and is assumed to occur *instantaneously.*

Other hardware-specific details that are assumed by each protocol will be mentioned in the protocols section below.

Protocols

We implement the following protocols:

1. MESI
2. MOESI
3. MESIF
4. Dragon

MESI

The MESI protocol introduces 4 states to a cache line: M(odified), E(xclusive), S(hared) and I(nvalid). There are many different variations of MESI; our implementation tries to follow the Illinois protocol, which is an *extended* form of MESI as it allows *clean-sharing* i.e. a cache that has the requested line in the E or S state can respond to the request while inhibiting the main memory from responding. This is not possible in simple MESI as matching caches in the S state would not know who should be responsible for sending the data. The Illinois protocol assumes a hardware-based daisy chain which enables a unique sender. However, such a daisy chain would incur cycle costs; we model this by assuming that the cost is (P + 1) cycles, where P is the number of processors.

The MESI protocol implemented in our simulator is thus as follows:

1. On Read Miss:
    a. A *BusRd* transaction is broadcasted to all caches and main memory
    b. If any other cache has the line, the main memory is inhibited from responding to the request
    c. If at least one other cache has the data, exactly one cache will do a cache-to-cache transfer
    d. All caches that have the line transitions to the S state
    e. If the responding cache has the line in the M state, it transitions to the S state *while* writing back to memory and doing the cache-to-cache transfer
2. On Write Miss:
    a. A BusRdX transaction is broadcasted to all caches and main memory. This has the same effect as a *BusRd*, but also invalidates the receivers' cache line in the process.
    b. The response is handled similarly to a read miss, except if the receiver has the cache line, it transitions to the I state instead of the S state.
3. On Read Hit: No bus transactions are generated. The cache line remains in the same state.
4. On Write Hit:
    a. If the requesting cache has the line in the M state, then no writes to the main memory are performed. The cache line remains in the M state
    b. If the requesting cache has the line in the E state, then no writes to the main memory are performed. The cache line transitions to the M state.
    c. If the requesting cache line has the line in the S state, then an invalidation signal is asserted (no bus traffic is generated). The cache line transitions to the M state.

MOESI

The MOESI protocol we implement extends the Illinois protocol with *dirty-sharing*. An additional state O(wner) is added to the protocol. When a cache has the line in the M state but receives a read request, the cache transitions to the O state and responds to the request with the data **without** writing back to memory. Thus, writes to the main memory only occur when a cache line is evicted from the M state, with the owner cache being responsible for write-backs.

The MOESI protocol implemented in our simulator is thus as follows:

1. On Read Miss:
    a. This is similar to MESI
    b. If any other cache has the line and it is in the O or M state, it is responsible for sending the cache line. No daisy chain penalty is incurred and no writes to memory are done
    c. If any other cache has the line but nobody has it in the O or F state, the sending cache needs to be arbitrated via a daisy chain. Hence, a daisy chain penalty is incurred
2. On Write Miss:

    a. Responses are handled similarly to Read Miss, except instead of transitioning to S state, the responding caches transition to I state
3. On Read Hit: No bus transactions are generated. The cache line remains in the same state.
4. On Write Hit:
    a. If the cache line is in M or E state, then no writes to the main memory are performed. The cache line remains in the M state.
    b. If the cache line is in S or O state, then an invalidation signal is asserted (no bus traffic is generated). The cache line transitions to the M state.
5. On eviction, a cache line is written back to memory only if it is in the M state.

MESIF

The MESIF protocol we implement extends the Illinois protocol with *clean-sharing*. An additional state F(orward) is added to the protocol. When the processor reads miss on a cache line that is shared, it transitions to the Forward state. This allows clean-sharing amongst the processors without having to heavily rely on the daisy chain as the cache that is in the F state is always responsible for cache-to-cache transfer. Occasional use of the daisy chain is still expected. For example, when the cache line is shared but no cache has it in the F state (e.g. when the cache which previously have it in the F state evicts that line), and a read miss occurs, the daisy chain is used to decide which cache does the transfer. In this case, the daisy chain penalty is incurred on top of the cache-to-cache transfer penalty.

The MESIF protocol implemented in our simulator is thus as follows:

1. On Read Miss:
    a. This is similar to MESI
    b. If any other cache has the line and it is in the F state, it is responsible for sending the cache line. No daisy chain penalty is incurred
    c. If any other cache has the line but nobody has it in the F state, the sending cache needs to be arbitrated via a daisy chain. Hence, a daisy chain penalty is incurred
2. On Write Miss:
    a. Responses are handled similarly to Read Miss, except instead of transitioning to S state, the responding caches transition to I state
3. On Read Hit: No bus transactions are generated. The cache line remains in the same state.
4. On Write Hit:
    a. If the cache line is in M or E state, then no writes to the main memory are performed. The cache line remains in the M state.
    b. If the cache line is in S or F state, then an invalidation signal is asserted (no bus traffic is generated). The cache line transitions to the M state.
5. On eviction, a cache line is written back to memory only if it is in the M state.

Dragon

The Dragon Protocol has 4 states - Modified(M), Shared-Modified(Sm), Shared-Clean(Sc) and Exclusive(E). Modified and Exclusive states in the Dragon protocol are similar to those of the MESI protocols and their variants. The Dragon Protocol is an update-based protocol adapted from the Xerox PARC Dragon processor.

The bus transactions are Bus Update and Bus Read. The processor transactions are Processor read and processor write.

The behaviour is as follows:

1. On Read Miss:
    a. A *BusRd* bus transaction is broadcasted to all caches and main memory
    b. If any other cache has the line, the main memory is inhibited from responding to the request
    c. If a cache line is shared, the cache line within the processor transitions to Sc and the responding cache line either transitions from M->Sm or E->Sc or remains as Sc.
    d. If a cache line is not shared, the cache line within the processor transitions to E.
2. On Write Miss:
    a. A *BusRd* bus transaction is broadcasted to all caches and main memory
    b. If the cache line is shared, the cache line within the processor transitions to Sm. A *BusUpd* is then sent to all shared cache.
    c. If the cache line is not shared, the cache line within the processor transitions to M.
3. On Read Hit: No bus transactions are generated. The cache line remains in the same state.
4. On Write Hit:
    a. If the cache line is in the M state, then no writes to the main memory are performed. The cache line remains in the M state.
    b. If the cache line is in the E state, then no writes to the main memory is performed. The cache line transitions to the M state.
    c. If the cache line is in Sm state, A `BusUpd` is then sent to all shared cache. The cache line remains in Sm.
    d. If the cache line is in Sc state, A `BusUpd` is then sent to all shared cache. The cache line transitons to Sm.
5. On eviction, a cache line is written back to memory if it is in the M or Sm state.

Implementation Detail

At initialisation, we create a shared bus, the processors and their respective caches, as well as a memory controller. The simulator is *single-threaded* and the simulation runs on a per-cycle basis. At each cycle, each processor is triggered once. To suspend execution e.g. due to an instruction taking more than one cycle, etc. the processors would save its state and return

control back to the caller. This approach is used heavily throughout the code base. While this is not ideal, this approach is more easily understood than having to use C++ co-routines.

For memory read/writes, the processor must first acquire the bus before anything can be done. The acquire process will try to register the requesting processor if it has not been registered and will return either the current ID of the current bus owner or the “next-in-line” processor waiting for the bus. The processor then either continues execution if the returned processor ID is itself, or suspends execution otherwise. In this way, FIFO arbitration for the bus is achieved.

Due to the single-threaded nature of the simulator, when a processor posts a request onto the bus, the processor must poll all other processors for the other caches to be able to respond to the request. All processors maintain their own “response” states i.e. the responding processor will not send its response if it is not the time yet. In this way, the responding processors can “force” the requesting processor to stall and suspend its execution with the correct number of cycles. When the requesting processor is unable to proceed, it will suspend its execution by returning control *without* releasing the bus. This maintains the bus’ atomicity. The next processor can then be run; if it looks to do read/write operations, it will not be able to do so since the bus has not been released yet. On the other hand, if it seeks to do a compute operation, it will be able to do so without being blocked.

A minor optimisation can be done on read hits; if a processor attempts to read data that is already present in its cache, then it can do so without having to acquire the bus. This relaxes the atomiticity of the bus slightly, but without compromising correctness.

A similar optimisation can be done on write hits, but doing so will lead to incorrect data being sent on cache-to-cache transfer, and possibly incorrect state transitions in certain cases. Hence, we require write hits to wait on the bus even if the data is in the cache.

Program Environment

We implemented the cache simulator in C++. Some C++20 features are used; hence, compilers supporting C++20 features are needed to build the project. We have tested this codebase with GCC 11.4.0, GCC 12.3.0, and Clang 15.0.7 on Ubuntu 22.04 and Ubuntu 18.04. There are known issues with MSVC due to its incomplete support for C++20.

This codebase also uses CMake>3.20 as we use the `FetchContent` module to download the *argparse* header-only library which is used for parsing command line arguments.

Results and Analysis

All results were collected with the cache configuration of: 4096KB cache size, 2 way set associativity, 32 Byte cache line size, byte addressable memory.

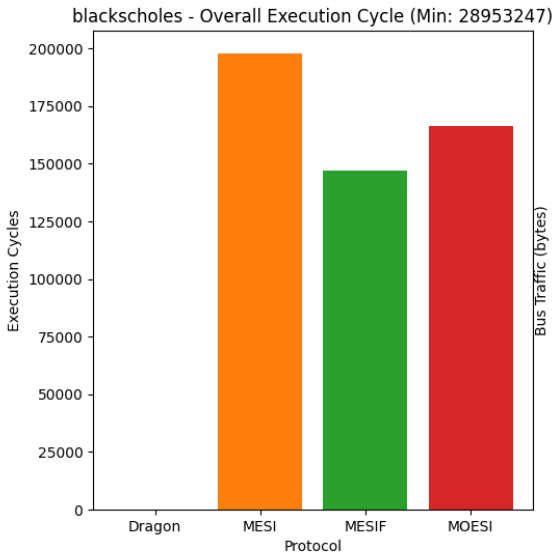
Benchmarks

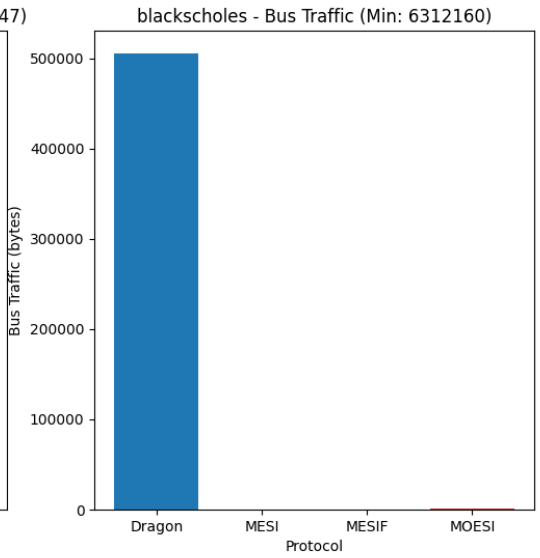
Number of cycles:

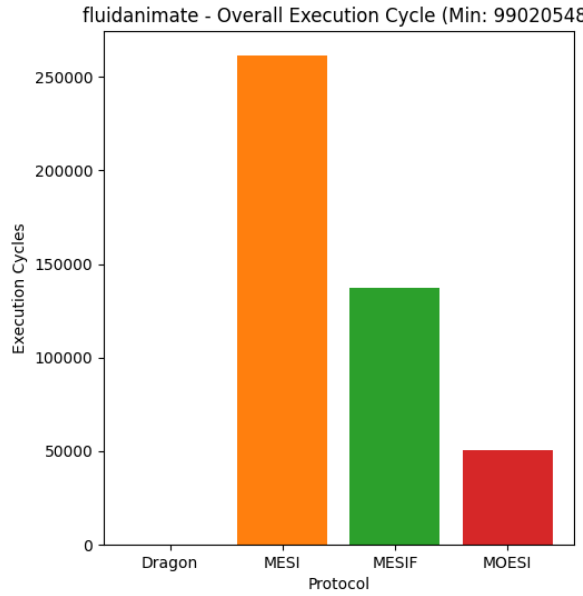
Protocol	**Black-Scholes**	**Fluid Animate**	**Body-Track**
MESI	29151141	99281981	69597708
MOESI	29119565	99071101	69461801
MESIF	29100447	99157820	69357350
Dragon	28953247	99020548	69177630

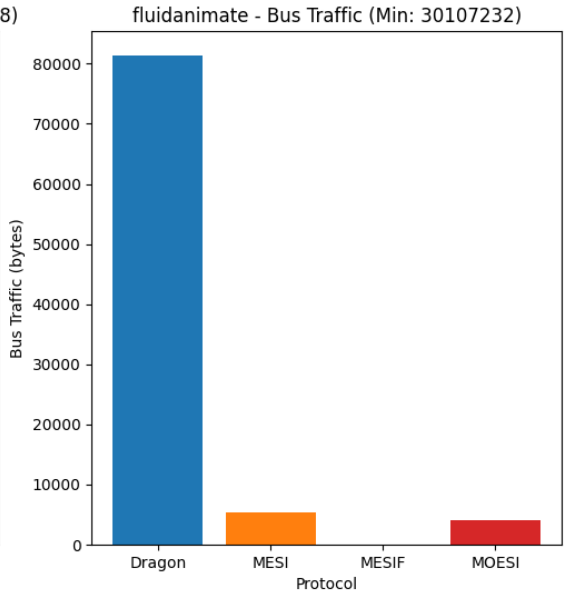
Bus Traffic (Bytes):

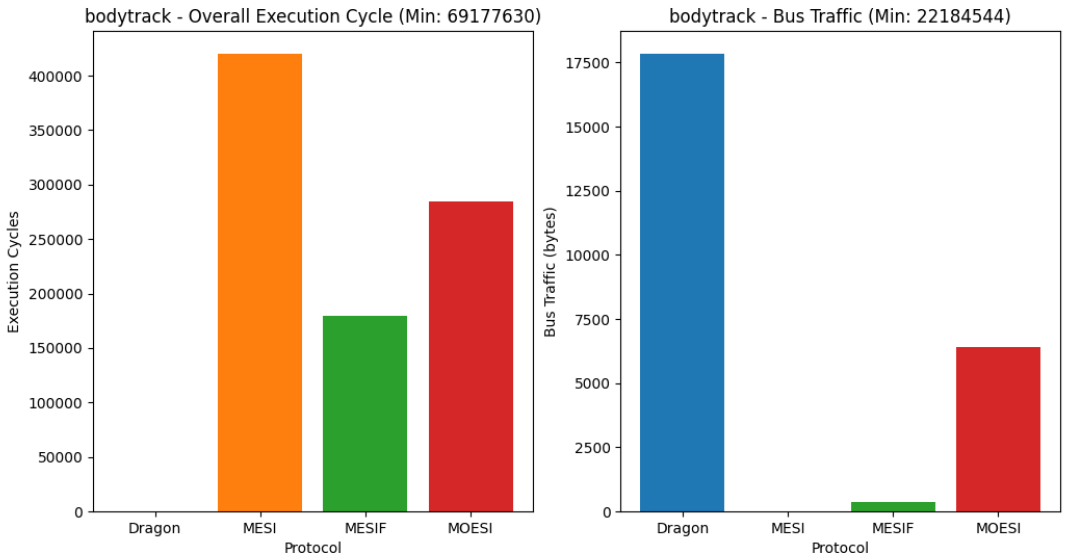
Protocol	**Black-Scholes**	**Fluid Animate**	**Body-Track**
MESI	6312160	30112608	22184544
MOESI	6312800	30111264	22190944
MESIF	6312160	30107232	22184896
Dragon	6817668	30188604	22202388


blackscholes - Overall Execution Cycle (Min: 28953247)
Execution Cycles
200000
175000
150000
125000
100000
75000
50000
25000
0
Dragon
MESI
MESIF
MOESI
Protocol


blackscholes - Bus Traffic (Min: 6312160)
Bus Traffic (bytes)
500000
400000
300000
200000
100000
0
Dragon
MESI
MESIF
MOESI
Protocol


fluidanimate - Overall Execution Cycle (Min: 99020548)
Execution Cycles
250000
200000
150000
100000
50000
0
Dragon
MESI
MESIF
MOESI
Protocol


fluidanimate - Bus Traffic (Min: 30107232)
Bus Traffic (bytes)
80000
70000
60000
50000
40000
30000
20000
10000
0
Dragon
MESI
MESIF
MOESI
Protocol



Observed trends on the benchmarks:

1. Dragon has the lowest execution cycle for all three benchmarks, but also the highest bus traffic
2. Among all invalidation protocols, MESI performs the worst in terms execution cycle, but the best in terms of bus traffic
3. MESIF is faster than MOESI in Black-Scholes and Bodytrack, but not in Fluidanimate:
    a. This might be due to more opportunity for dirty-sharing in Fluidanimate than in the other benchmarks
4. MESIF has lower bus transactions than MOESI in all 3 benchmarks

Custom Test Cases

Note that for all of the following test cases:

- There are 1 million instructions per core
- All instructions are memory operations operating on the same memory location

In this way, these test cases serve as possible worst-case scenarios for a cache-coherence system.

Number of Cycles

Protocol	All Writes	Random read-writes	1 write many reads
MESI	399999999	200007370	2000380
MOESI	64000083	32002074	2000212
MESIF	399999999	200007370	2000360
Dragon	8000142	4030359	2000145

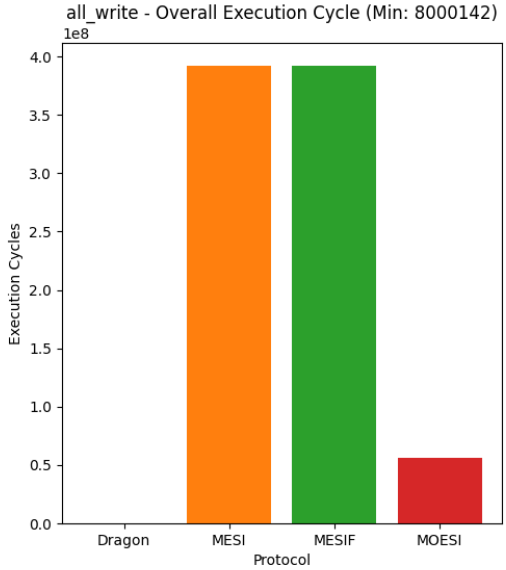
Bus traffic (Bytes)

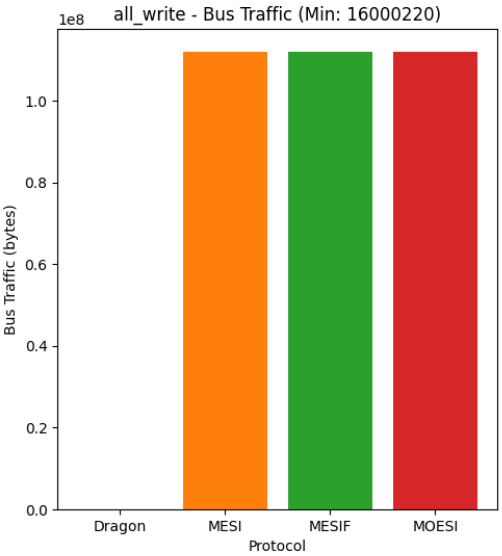
Protocol	All Writes	Random read-writes	1 write many reads
MESI	128000000	64002336	224
MOESI	128000000	64003296	224
MESIF	128000000	64002336	224
Dragon	16000220	8000964	4000124

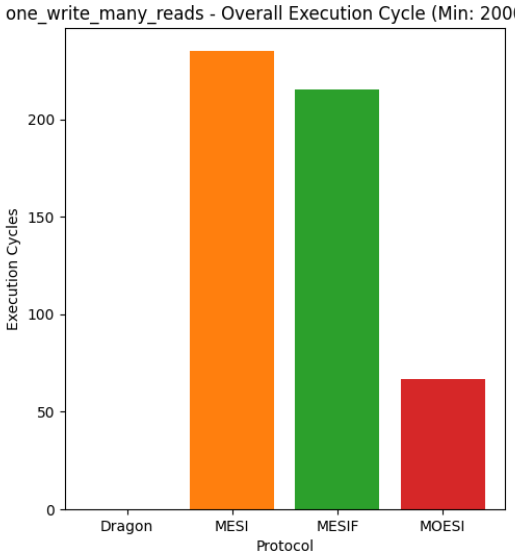
All Writes
As expected, Dragon performs the best as it updates all the shared cache lines during each write. MOESI is the next best due to the presence of the O state. This allows dirty-sharing, which prevents writing back to main memory.

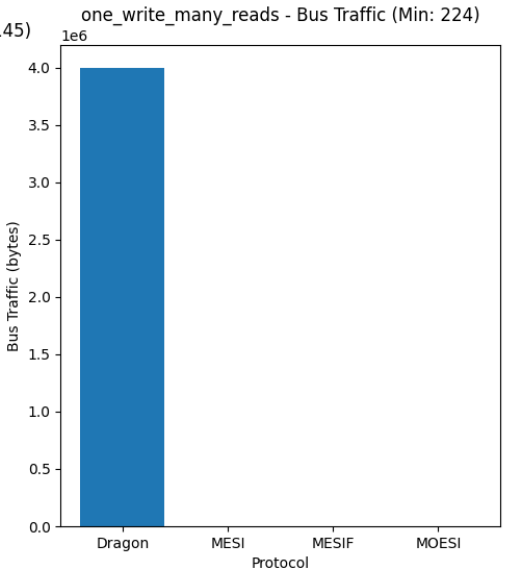
Random read-writes
Dragon once again performs the best. MOESI is the best due to the ability to do dirty sharing. MESI and MESIF perform similarly.

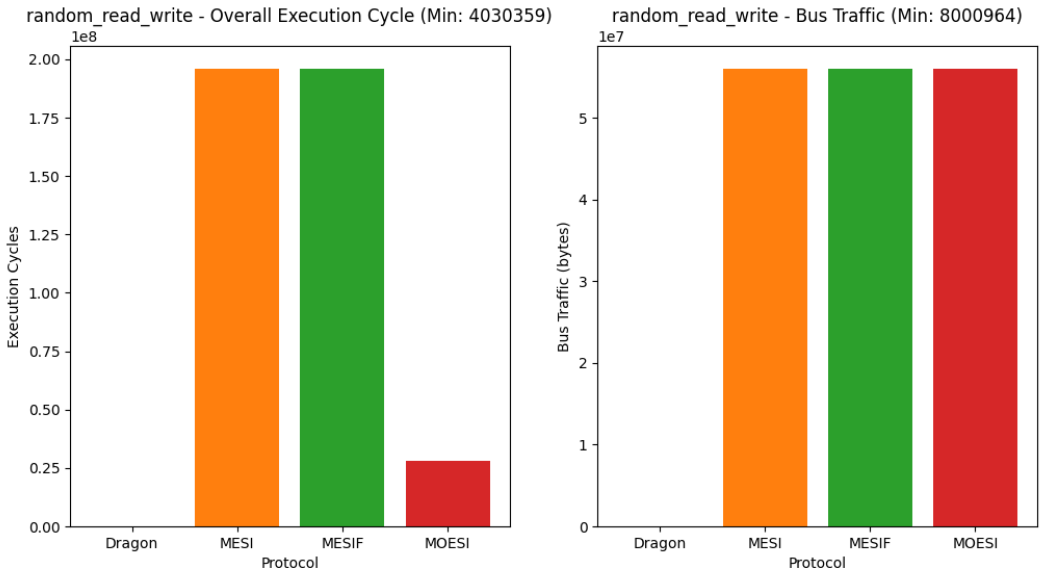
One write Many reads
In this case Processor 0 only performs writes to the same location in memory. All other processors read from the same location in memory. Dragon performs the best again however, it generates alot more bus traffic compared to the MESI variants. This is due to the need to do a bus update when P0 does a write. However, in the MESI variants, many write hits occur and no update is done.


all_write - Overall Execution Cycle (Min: 8000142)
1e8
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Execution Cycles
Dragon
MESI
MESIF
MOESI
Protocol


all_write - Bus Traffic (Min: 16000220)
1e8
1.0
0.8
0.6
0.4
0.2
0.0
Bus Traffic (bytes)
Dragon
MESI
MESIF
MOESI
Protocol


one_write_many_reads - Overall Execution Cycle (Min: 2000145)
200
150
100
50
0
Execution Cycles
Dragon
MESI
MESIF
MOESI
Protocol


one_write_many_reads - Bus Traffic (Min: 224)
1e6
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Bus Traffic (bytes)
Dragon
MESI
MESIF
MOESI
Protocol



Optimization (write buffer)

We implemented a write buffer as an optimization for the system. During write backs, the cache line is sent to the write back buffer which is modelled as another cache. Writes and reads from the write back buffer takes 2*N cycles, where N is the number of words per cache line. The write buffer has no limit.

Benchmarks

Number of cycles:

Protocol	**Black-Scholes**	
	No Buffer	**With Buffer**
MESI	29151141	23322161
MOESI	29119565	23308112
MESIF	29100447	23233020
Dragon	28953247	23261389

Number of cycles:

Protocol	Fluid Animate	
	No Buffer	**With Buffer**
MESI	99281981	42042497
MOESI	99071101	42057621
MESIF	99157820	41979165
Dragon	99020548	41823813

Number of cycles:

Protocol	Body-Track	
	No Buffer	**With Buffer**
MESI	69597708	58215103
MOESI	69461801	58743086
MESIF	69357350	57574325
Dragon	69177630	58104262

As expected, in all protocols, the addition of a write back buffer leads to a lower number of cycles for each test set. This is expected as the delays for write backs is cut down from 100 cycles to 2*N cycles.

An interesting observation we see is that Dragon and MOESI do not benefit as much as MESI and MESIF when the write back buffer is added. This could be because MESI and MESIF has more write backs to memory and hence benefit from it more.

Custom Test Cases

Number of cycles:

Protocol	All Writes	
	No Buffer	**With Buffer**
MESI	399999999	64000083
MOESI	64000083	64000083
MESIF	399999999	64000083
Dragon	8000142	8000142

Number of cycles:

Protocol	Random Read Writes	
	No Buffer	**With Buffer**
MESI	99281981	32002074
MOESI	99071101	32002074
MESIF	99157820	32002074
Dragon	99020548	4030359

Number of cycles:

Protocol	One Write Many Reads	
	No Buffer	**With Buffer**
MESI	2000380	2000212
MOESI	2000212	2000212
MESIF	2000360	2000192
Dragon	2000145	2000145

We observer similar trends as the benchmarks; MESI and MESIF seems to benefit the most from the addition of write back buffers, while MOESI benefits to a lesser extent and Dragon to the smallest extent.

Conclusion

From our experiments, Dragon performs the best compared to all other protocols in terms of the number of cycles taken to complete the test set. However, it generates more bus traffic compared to the rest. The MESI variants perform somewhat similarly in all test cases. MESI performs the worst out of the 3 as expected as it is not optimised in any way. MOESI does better in fluid animate and this could be due to the presence of more dirty sharing. MESIF performs better in the other 2 test sets (black scholes and body track). This could be due to the presence of more clean sharing.

We also found that the cache coherence protocol performances are highly affected by the underlying hardware architecture and thus how it is modelled in the simulator e.g. presence write back buffers, presence of daisy chain hardware and its latency, etc. This explains some real-world phenomena, such as how Intel CPUs tend to use MESIF and AMD CPUs tend to use MOESI; their design decisions are highly likely to be dictated by the underlying hardware design of their chip such as the bus.