CS4223: Multi-Core Architectures

Mini-Project report

Team members:

- Wu Qihao, A0154900M, e0031037@u.nus.edu
- Nima Alikhani, A0225355H, nima.alikhani@u.nus.edu

Table of contents :

Introduction

In the CS4223 course, we have seen multiple cache coherence protocols. These protocols were of 2 global types: the write invalidate protocol and the write update protocol.
The invalidation ones work by invalidating cache lines when a write is observed, snooped, in another cache. The examples we have seen in lessons are MSI and MESI. However, in tutorials, we have also seen MOESI and MESIF, which are AMD and Intel enhancement of the previous protocols.
The write update protocol works by updating cache lines after write operations. An example we have seen is Dragon.

One may ask how is cache coherence protocol better. The fact is that it depends on the situation. The cache and the type of programs are involved in this choice. This is what this assignment is about. Thanks to a trace-based cache simulator, we will examine 3 benchmarks and see how the different protocols are competing with each other.

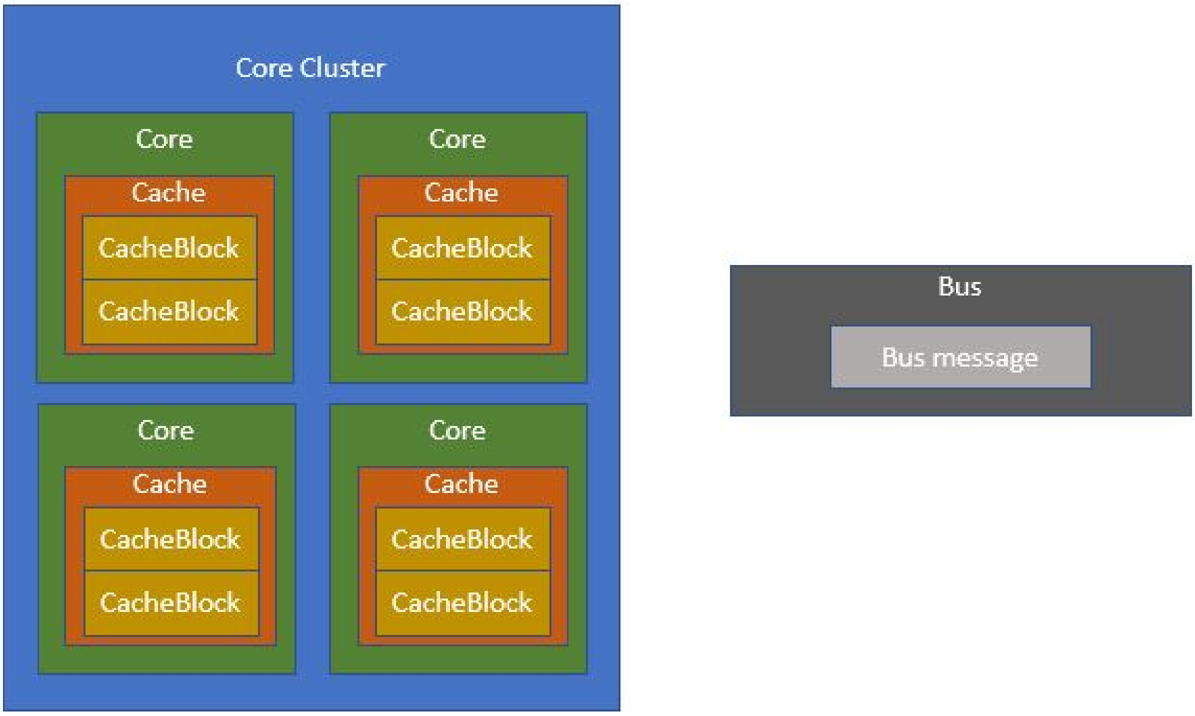
Programming Language and Environment

The programming language choice was an essential step. Specific programming paradigms can simplify a programmer's life.
After examining the assignment, we decided to choose an Object-Oriented language. Indeed, as we are dealing with hardware simulators, implementing the different parts in objects (the bus or the cache for example) is easier.
Giving our team experience, we had to choose between C++ and Java. We both voted for C++ as it is at a lower level, which allows more efficiency.
Our programming language is C++ in its 17 standard (C++17).

The environment we used to program is **CLion**. It is an IDE developed by Jetbrains free for students. It automatically generates a CMake file to compile the project, which is useful.

Implementation

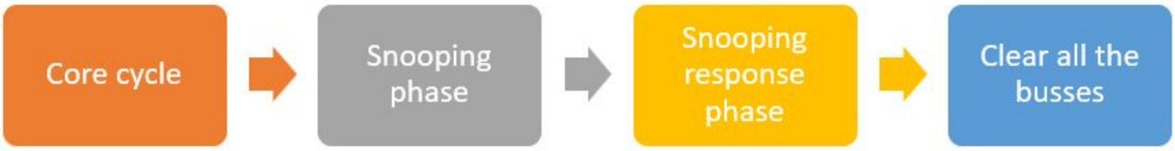
Let's first take a bird's eye view of all the implemented classes.



We implemented several classes that simulate the real hardware components.

- The Core Cluster is a container for all the Cores. It is the interface directly provided to the programmer to use all the cores. It is responsible for running the program until all cores have finished.
- The Core is the class that runs the program. Each core has its Cache.
- The Cache is where cache coherence is implemented. It is made of the cache itself and all the snooping parts.
- The CacheBlock is the class representing a cache line/block.
- The Bus is a class that allows Cache to communicate with each other through BusMessage. Our implementation features 2 busses: the main bus and the response bus. We will see the details later on.
- The BusMessage is the class representing a message on the bus.
-

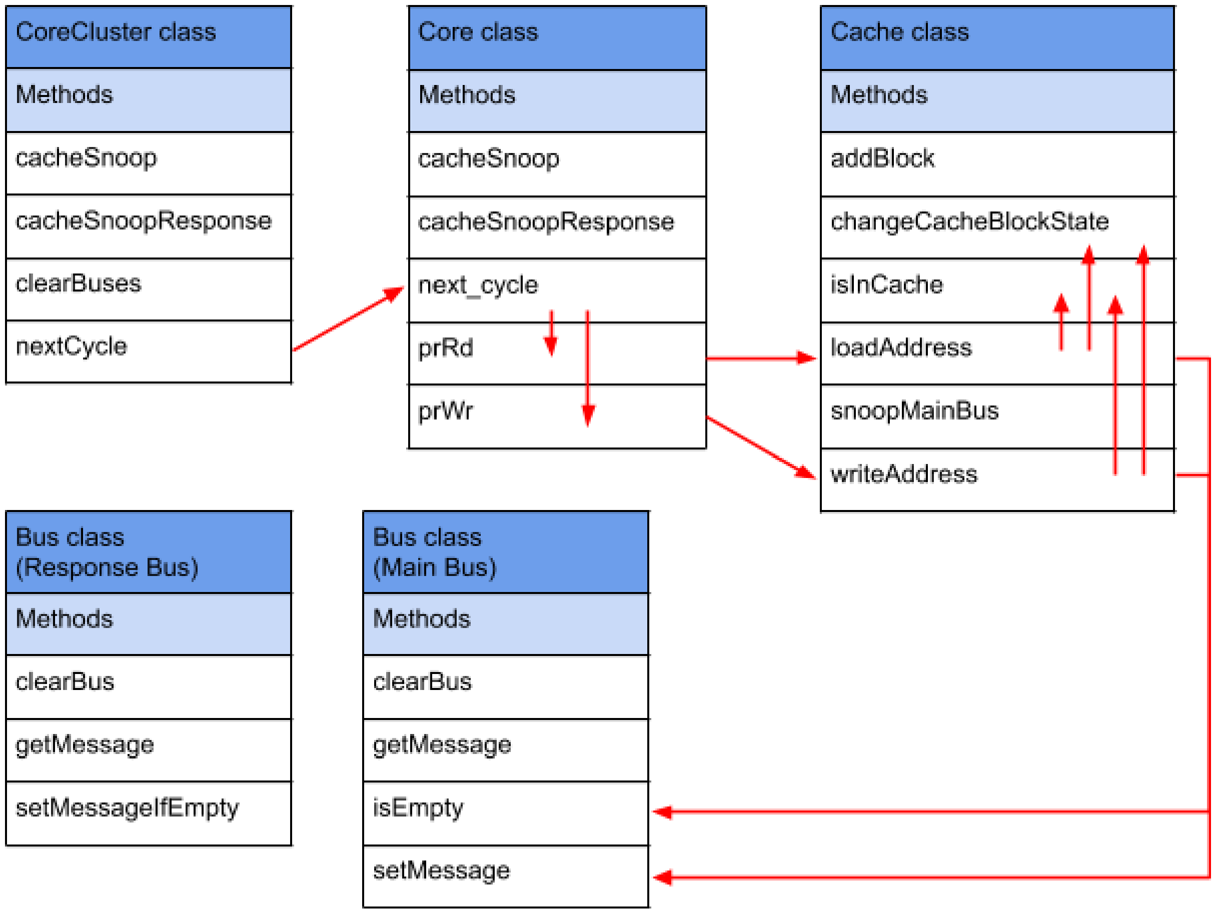
The overall diagram is at the end of the report.
What is important to understand about our implementation is that each cycle requires 4 steps.



Let us explain what each step is about.

Step 1: each core proceed with its next cycle if possible (if not blocked by cache or program end)

The called methods are pointed by arrays on this chart.



CoreCluster class
Methods
cacheSnoop
cacheSnoopResponse
clearBuses
nextCycle

Core class
Methods
cacheSnoop
cacheSnoopResponse
next_cycle
prRd
prWr

Cache class
Methods
addBlock
changeCacheBlockState
isInCache
loadAddress
snoopMainBus
writeAddress

Bus class (Response Bus)
Methods
clearBus
getMessage
setMessageIfEmpty

Bus class (Main Bus)
Methods
clearBus
getMessage
isEmpty
setMessage

The core class may proceed with a processorRead (prRd) or a processorWrite (prWr) if not blocked or in a computing cycle. It transmits its order to the cache. The cache may check if the corresponding cache block is in the cache, and possibly change its state.

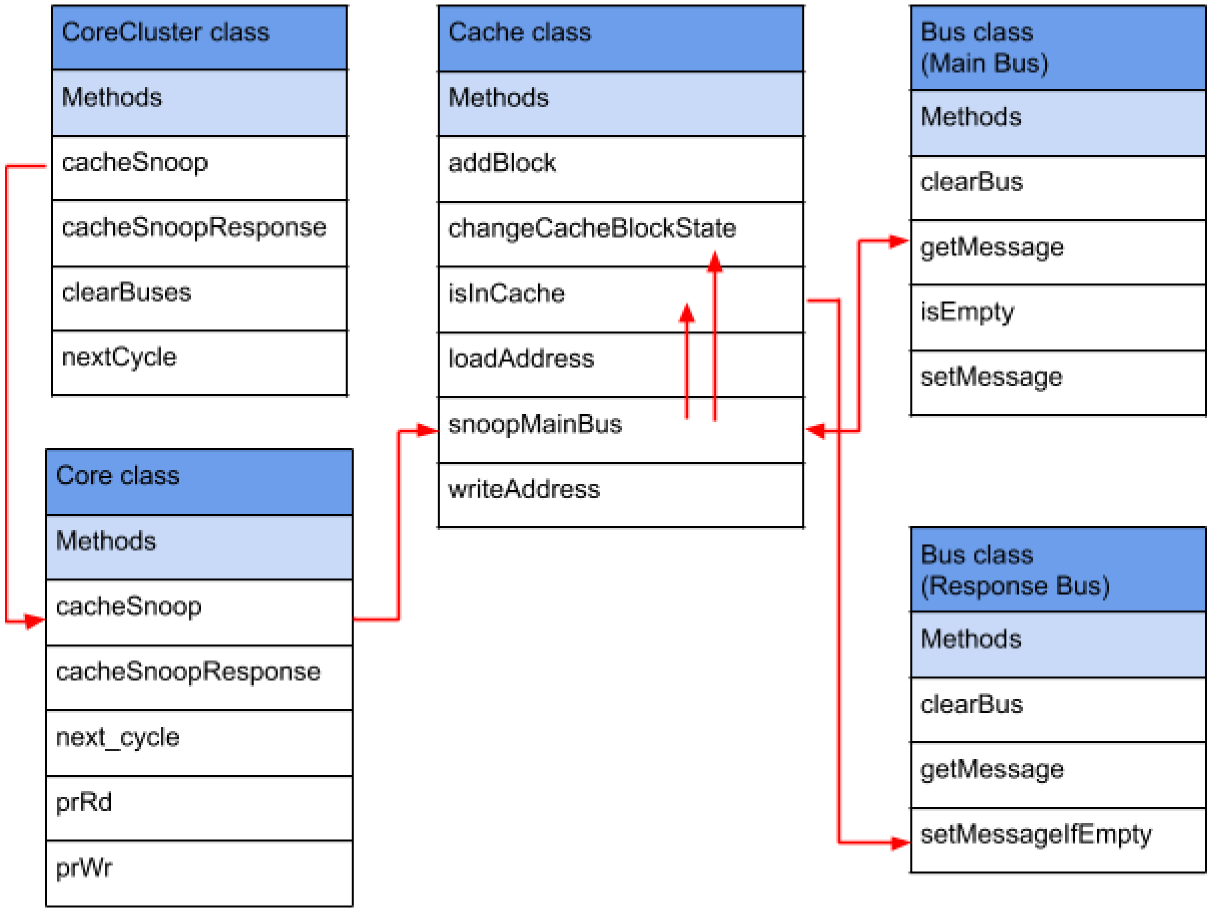
However, in some situations, it may require information about other caches, i.e. if another cache has the block or even updates. As an alternative to a shared line, provide more versatility, a structure with two busses is implemented.

In this step, one core writes a transaction on the Main Bus. We will later see in steps 2 and 3 how the second bus, the Response Bus, comes into play.

Note that only one cache is allowed to post a message on the bus. If the main bus is not available, the core waits until it is.

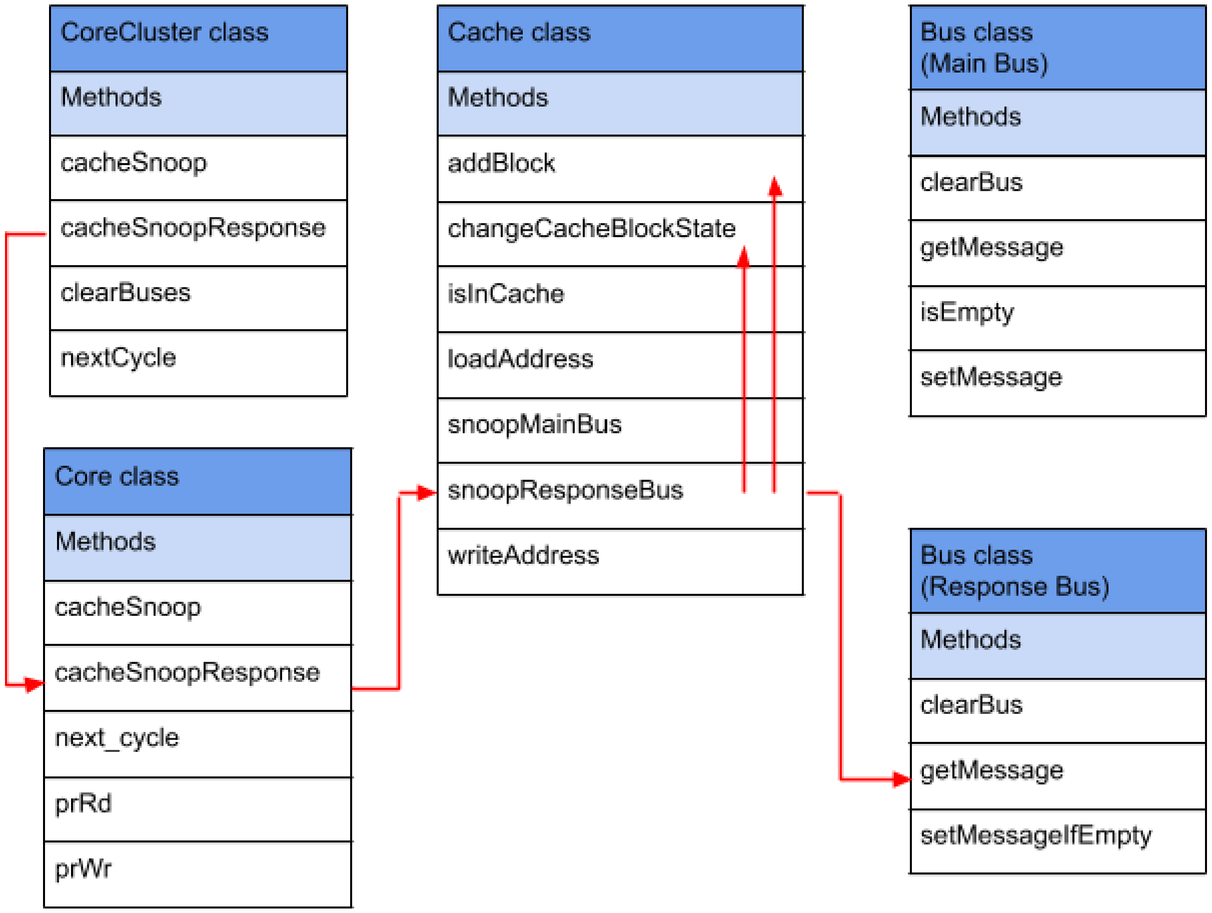
Step 2: each core snoops on the main bus to update their cache and, if necessary, put a message on the response bus

Here, each cache is allowed to change their cache blocks state according to the current bus transaction on the main bus, but also to react to a bus transaction. The cache might, for example, indicate that it has a cache block inside or even transmit this cache block.

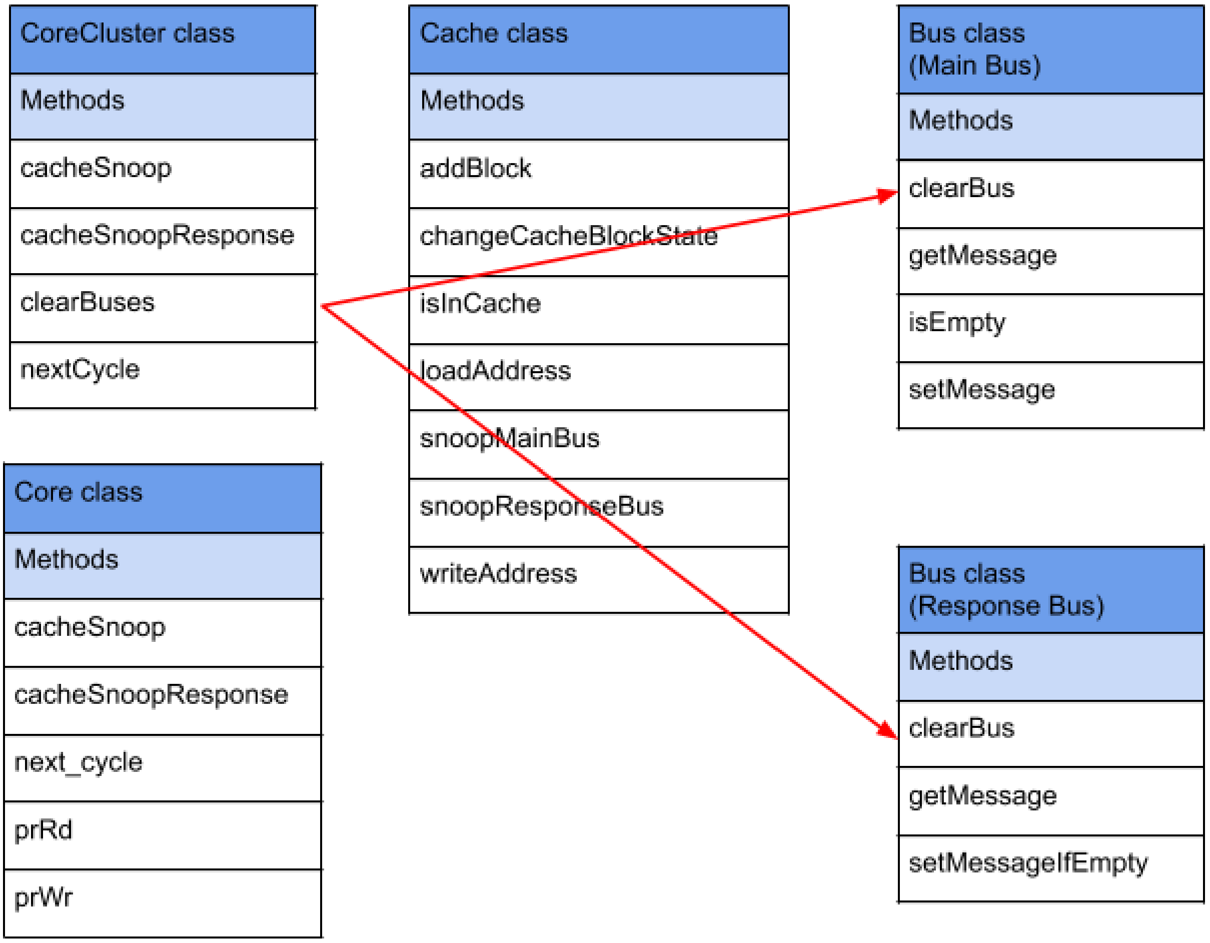


Step 3: the core which has posted the message on the main bus snoop the response bus, and if possible, react.

After the snooping phase, we left the opportunity to the core which posted the message on the main bus to react to it. It may be by changing a cache state or by changing the number of cycles to wait for a specific operation.



Step 4: clear the 2 buses to prepare the next cycle



CoreCluster class
Methods
cacheSnoop
cacheSnoopResponse
clearBuses
nextCycle

Cache class
Methods
addBlock
changeCacheBlockState
isInCache
loadAddress
snoopMainBus
snoopResponseBus
writeAddress

Bus class (Main Bus)
Methods
clearBus
getMessage
isEmpty
setMessage

Core class
Methods
cacheSnoop
cacheSnoopResponse
next_cycle
prRd
prWr

Bus class (Response Bus)
Methods
clearBus
getMessage
setMessageIfEmpty

Protocols and Transitions:

MESI

We will describe the MESI protocols thanks to 2 tables we found on Wikipedia[1]. Of course, this description will be accurate implementation-wise, this is the specification we aimed to implement, so there are modifications made by ourselves.

Intuitively, the programs that will take advantage of this invalidation protocol are those where there are many writes to the same cache block. In an update protocol (like Dragon), updates will need to be issued at each write, that's not the case here because the invalidation is only issued once.

Please note that here, the shared line was implemented thanks to the 2 busses.

Table 1 is a description of the protocol in steps 1 and 3. Table 2 is the description in step 2.

Table 1: response to core operations during step 1

Initial State	Operation	Response
Invalid(I)	PrRd	● Issue BusRd on the main bus, wait for step 3 to have a response on the response bus ○ If there is one, transition to Shared, cache-to-cache transfer (in step 3) ○ If none, transition to Exclusive, fetch from main memory (in step 3)
	PrWr	● Issue BusRdX on the main bus, wait for step 3 to have a response on the response bus ○ If there is one, transition to Shared, cache-to-cache transfer (in step 3) ○ If none, transition to Exclusive, fetch from main memory (in step 3) ● Transition to Modified
Exclusive(E)	PrRd	● Direct cache hit
	PrWr	● Transition to Modified ● Direct Cache Hit

[1] https://en.wikipedia.org/wiki/MESI_protocol

Shared(S)	PrRd	● Direct Cache Hit
	PrWr	● Issues BusUpgr on the main bus. ● Transition to Modified
Modified(M)	PrRd	● Direct Cache Hit
	PrWr	● Direct Cache Hit

Table 2: response to bus transactions during step 2

Initial State	**Operation**	**Response**
Exclusive(E)	BusRd	● Transition to Shared ● Put FlushOpt on the response bus with the block content (here, we do not have the data, so no real content)
	BusRdX	● Transition to Invalid. ● Put FlushOpt on the response bus with the block content.
Shared(S)	BusRd	● Put FlushOpt on the response bus with the block content (the first to do it win, only one message on a bus)
	BusRdX	● Transition to Invalid ● Put FlushOpt on the response bus with the block content
Modified(M)	BusRd	● Transition to Shared ● Put FlushOpt on the response bus with the block content. As it is also written to main memory, additional cycles are added to the corresponding core

	BusRdX	- Transition to Invalid - Put FlushOpt on the response bus with the block content. As it is also written to main memory, additional cycles are added to the corresponding core

MOESI

We will describe the MOESI protocols by highlighting the difference with MESI. The implementation differences are highlighted in yellow in the tables.

MOESI introduces a fifth state: the Owned State. This state spares some cycles when a cache tries to read a modified block of another cache. Instead of writing it back to the main memory, the data is directly transferred to the other processor via the bus.

Intuitively, the programs benefiting from this enhancement are those where one core writes data to a cache block, and the other cores will then attempt subsequent reads on the same cache block, to perform other calculations for example.

Table 3 is a description of the protocol in steps 1 and 3. Table 4 is the description in step 2.

Table 3: response to core operations during step 1 and 3

Initial State	Operation	Response
Invalid(I)	PrRd	- Issue BusRd on the main bus, wait for step 3 to have a response on the response bus ○ If there is one, transition to Shared, cache-to-cache transfer (in step 3) ○ If none, transition to Exclusive, fetch from main memory (in step 3)
	PrWr	- Issue BusRdX on the main bus, wait for step 3 to have a response on the response bus ○ If there is one, transition to Shared, cache-to-cache transfer (in step 3) ○ If none, transition to Exclusive, fetch from main memory (in step 3) - Transition to Modified
Exclusive(E)	PrRd	- Direct cache hit

	PrWr	● Transition to Modified ● Direct Cache Hit
Shared(S)	PrRd	● Direct Cache Hit
	PrWr	● Issues BusUpgr on the main bus. ● Transition to Modified
Modified(M)	PrRd	● Direct Cache Hit
	PrWr	● Direct Cache Hit
Owned(O)	PrRd	● Direct Cache Hit
	PrWr	● Issues BusUpgr on the main bus. ● Transition to Modified

Table 4: response to bus transactions during step 2

Initial State	Operation	Response
Exclusive(E)	BusRd	● Transition to Shared ● Put FlushOpt on the response bus with the block content (here, we do not have the data, so no real content)
	BusRdX	● Transition to Invalid. ● Put FlushOpt on the response bus with the block content.
Shared(S)	BusRd	● Put FlushOpt on the response bus with the block content (the first to do it win, only one message on a bus)

	BusRdX	● Transition to Invalid ● Put FlushOpt on the response bus with the block content
Modified(M)	BusRd	● Transition to Owned ● Put FlushOpt on the response bus with the block content.
	BusRdX	● Transition to Invalid ● Put FlushOpt on the response bus with the block content. As it is also written to main memory, additional cycles are added to the corresponding core
Owned(O)	BusRd	● Put FlushOpt on the response bus with the block content.
	BusRdX	● Transition to Invalid ● Put FlushOpt on the response bus with the block content. As it is also written to main memory, additional cycles are added to the corresponding core

DRAGON

We will describe the Dragon protocol by describing all its transitions[2]. This description will also be accurate implementation-wise, this is the specification we aimed to implement.

Intuitively, the programs that will take advantage of this update protocol are those with one write and subsequent reads on the same cache block. Indeed, as the write leads to an update, all the other caches will directly have the data available, and won't be fetching them.

Each cache block will be in one of the four states: **Exclusive-clean** (E), **Shared clean** (Sc), **Shared modified** (Sm), **Modify** (M). Unlike MESI, DRAGON protocol does not have invalid state. Instead, “invalid” state is signified by the absence.

A shared line is implemented to keep track of a cache block’s sharing status, whether it is available to multiple caches.

The following transactions form the description of the protocol in steps 1 and 3.

- When **prRdMiss** occurs, shared line assertion fail, state transition to **E**
- When **prRdMiss** occurs, shared line assertion pass, state transition to **Sc**
- When **prWrMiss** occurs, shared line assertion fail, state transition to **M**
- When **prWrMiss** occurs, shared line assertion pass, state transition to **Sm**
- When **prRd** hit, there will be no state change and no bus transaction
- When cache block in **M** state, and processor **prWr**, there will be no state transition as the block is not shared.
- When cache block in **Sm** state, and processor **prWr**, state transition to **M** if it is not shared
- When cache block in **Sm** state, and processor **prWr**, a **BusUpdate** will be generated if it is shared
- When cache block in **Sc** state, and processor **prWr**, state transition to **M** if it is not shared and no bus transaction will be generated
- When cache block in **Sc** state, and processor **prWr**, state transition to **Sm** if it is shared and **BusUpdate** will be generated
- When cache block in **E** state, and processor **prWr**, state transition to **M**

The following transactions form the description in step 2.

- When cache block in **M** state and **BusRd** is issued, **Flush** is issued to update the main memory, state transition to **Sm**
- When cache block in **Sm** state and **BusRd** is issued, **Flush** is issued to update the main memory, state remains the same
- When cache block in **Sm** state and **BusUpdate** is issued, state transition to **Sc** and all caches are updated
- When cache block in **Sc** state and **BusRd** or **BusUpdate** is issued, state remains the same. In the case of BusUpdate, it will update the cache block
- When cache block in **E** state and **BusRd** is issued, state transition to **Sc**, the blocked become shared

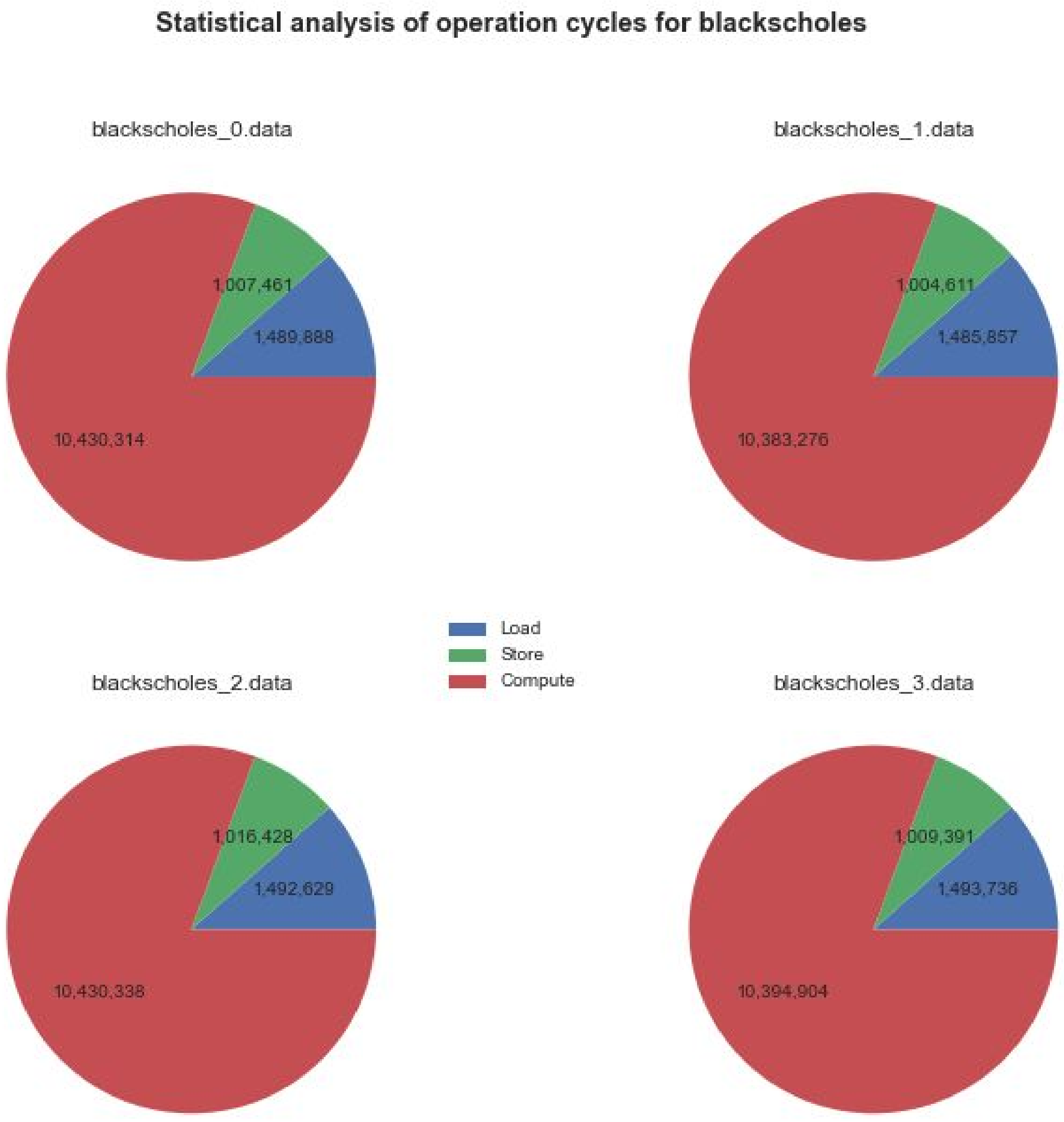
[2] https://en.wikipedia.org/wiki/Dragon_protocol

Experimental Results

Blackcholes

This benchmark trace is about a mathematical model used to calculate option pricing with Partial Differential Equations (PDE).

Let us first analyse the operations involved in this benchmark. In our analysis, we count the number of operation cycles. We suppose that every load and store counts as 1 cycle, whereas the computing cycles are the sum of all the compute cycle durations.



We notice two facts :

- all the 4 processors have similar statistics
- compared to the 2 other benchmarks, this one has a decent amount of loads and stores, even if they are only representing around ¼ of the cycles.

This last point let us foresee that any improvement in the load and store operations can have significant effects on the performance.

Optimizing MESI

In this part, we will try to find the best parameters for the MESI cache coherence protocol and this specific program. As we have done in assignment 1, we will proceed by testing some values but not all.

The cache parameters are the cache size, the associativity, and the block size. We will keep the cache size to 4kb and tweak the other parameters. We will aim to improve the overall execution cycle.

Block size / Associativity	16 bytes	32 bytes	64 bytes
1	111627159	116834523	92332073
2	40483996	45842296	51187182
8	26619002	29372802	45828147
16	27586520		
32	**26363213**		

We can see that the cache configuration giving the best result is here a 32-way set associative cache with 16 bytes block size. This is the configuration we will choose when comparing the protocols.

It seems that enhancing the block size is not that beneficial. The spatial locality benefice may be limited to this benchmark.

Optimizing MOESI

We will do the same method for MOESI :

Block size / Associativity	16 bytes	32 bytes	64 bytes
1	109311203	111695161	84718468
2	41959774	44764118	45649159
8	28785759	30382246	39991617

16	27575446		
32	**26357879**		

Optimizing DRAGON

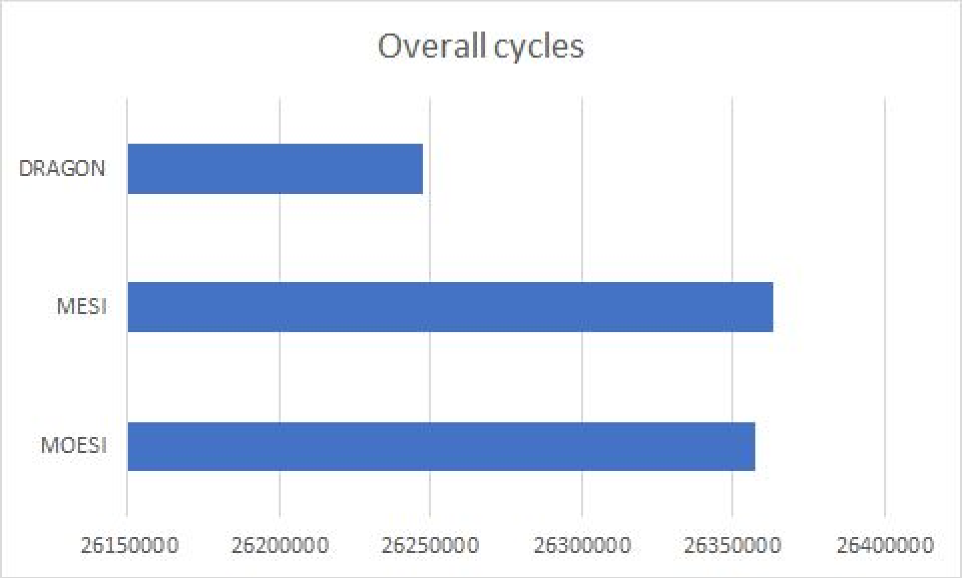
We will do the same method for DRAGON:

Block size / Associativity	16 bytes	32 bytes	64 bytes
1	79375029	80544813	65218722
2	40271712	42597123	41285967
8	29475861	30247962	37999860
16	27464676		
32	**26247768**		

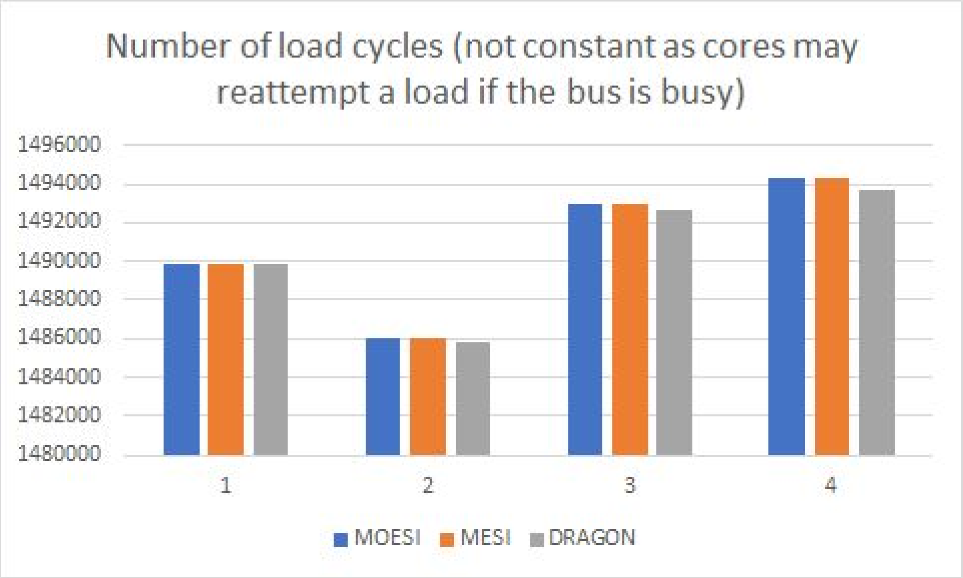
We can see that the cache configuration giving the best result is here a 32-way set associative cache with 16 bytes block size. This is the configuration we will choose when comparing the protocols.

Comparing the three protocols

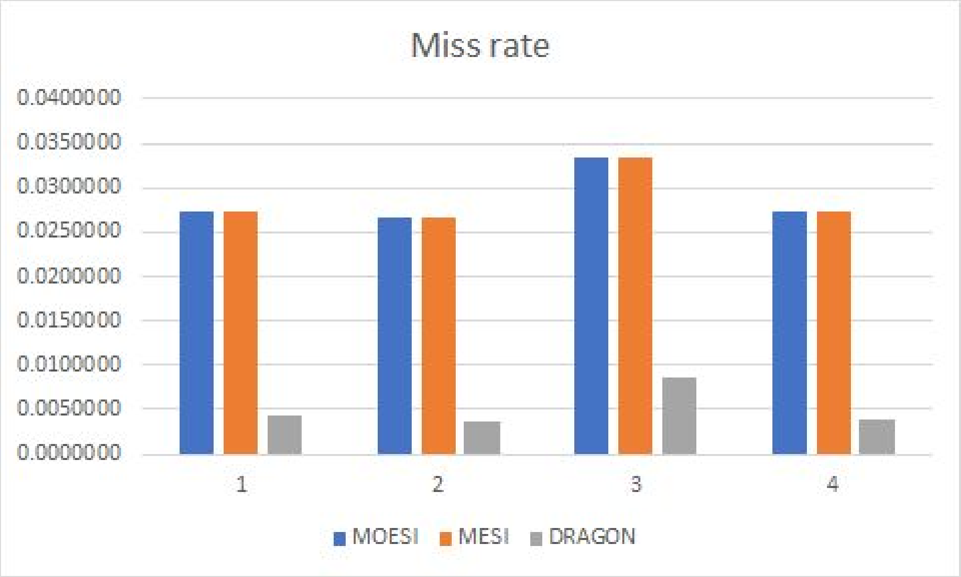
We have run the 3 simulations. We will analyze some of the relevant statistics.



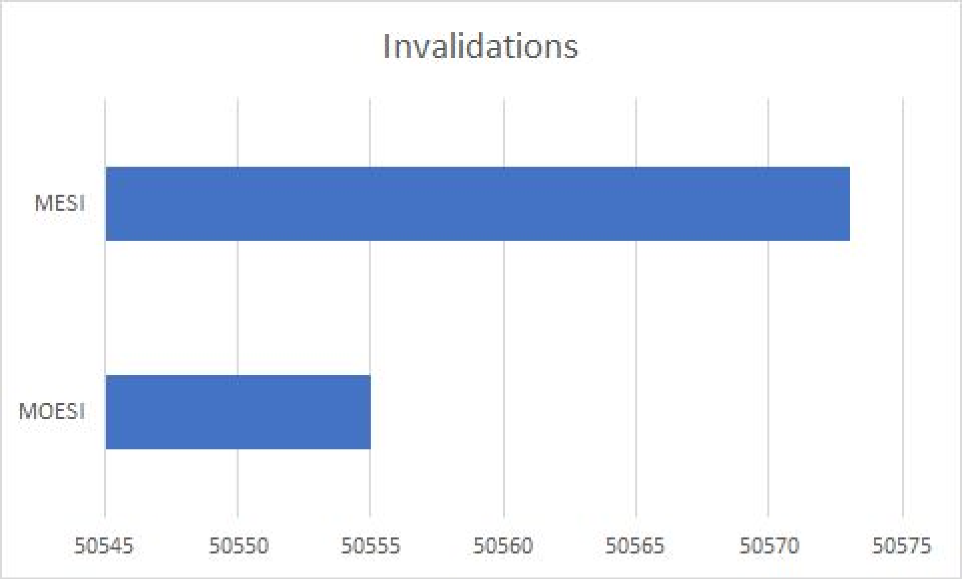
We can first observe that the Dragon protocol is the one allowing the fastest execution. However, as it has been observed in all simulations, we suspect it's due to a flaw in our implementation.



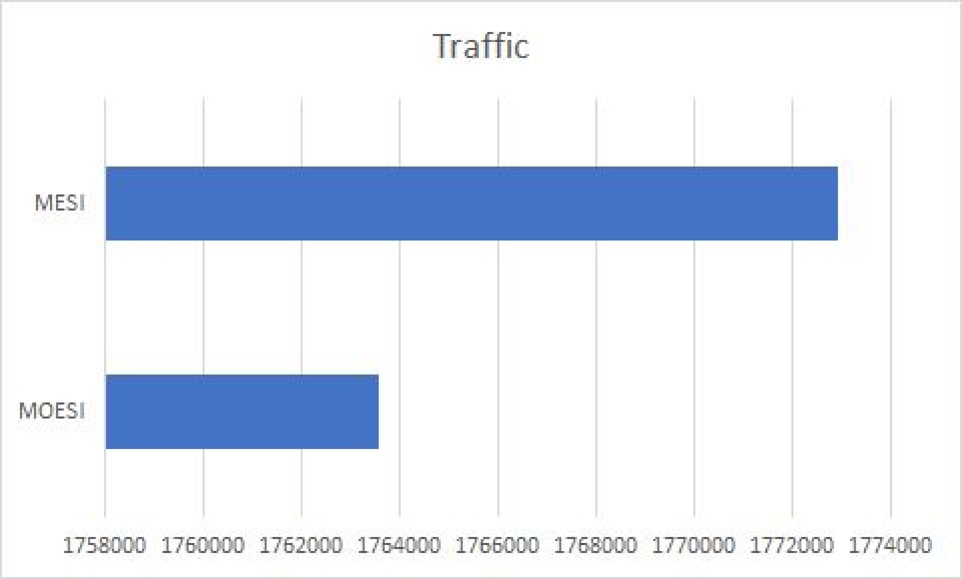
The number of load cycles reveals the heterogeneous number of loads across all cores. Moreover, implementation-wise, we can suspect that the high number of load cycles for the last cores is because of poor fairness of the bus allocation, as the first arrived is the first served.



The miss-rate shows us a major advantage of the Dragon protocol: it allows a significant reduction whereas the MOESI and MESI protocols are very close.



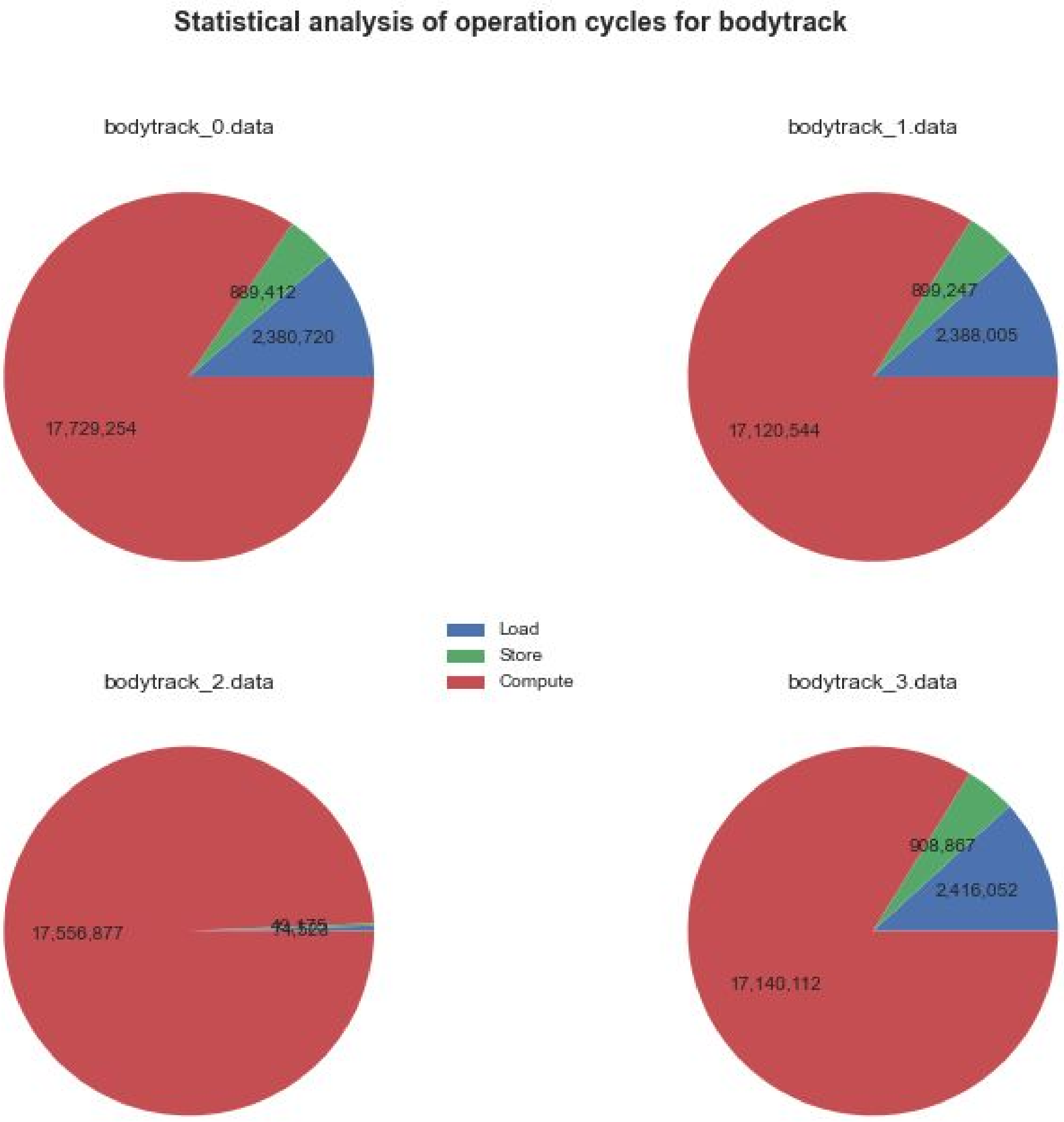
We can see that the number of invalidations is very close between the MESI and MOESI protocols (only ~25 more for MESI).



We can see that traffic is also close between our two invalidation protocols. Because of an implementation flaw, we were not able to get the DRAGON one. However, we would expect higher traffic as updates are constantly happening.

It seems that the DRAGON protocol is the best for this benchmark.

Bodytrack



We notice two facts :

- the third processor has far more memory operations, only the three others will do the majority of the memory loads and stores.
- compared to the 2 other benchmarks, this one has a higher number of memory transactions, and also the highest number of computing cycles.

Optimizing MESI

In this part, we will try to find the best parameters for the MESI cache coherence protocol and this specific program. The cache parameters are the cache size, the associativity, and the block size. We will keep the cache size to 4kb and tweak the other parameters. We will aim to improve the overall execution cycle.

Block size / Associativity	16 bytes	32 bytes	64 bytes	128 bytes
1	110100829	91592759	79648167	78363998
2	95181874	79494803	68598305	66752974
8	85756872	72264021	59629242	59501078
16	85982338			**58175550**
32	88872645			

We can see that the cache configuration giving the best result is here a 16-way set associative cache with 128 bytes block size. This is the configuration we will choose when comparing the protocols.

It seems that enhancing the block size is not that beneficial. The spatial locality benefice may be limited to this benchmark.

Optimizing MOESI

We will do the same method for MOESI :

Block size / Associativity	16 bytes	32 bytes	64 bytes	128 bytes
1	110271819	92336556	79672368	78331296
2	95155104	79378775	68554649	66666869
8	85702030	72208241	59662466	59493811
16	85921404			**58140491**
32	88801653			

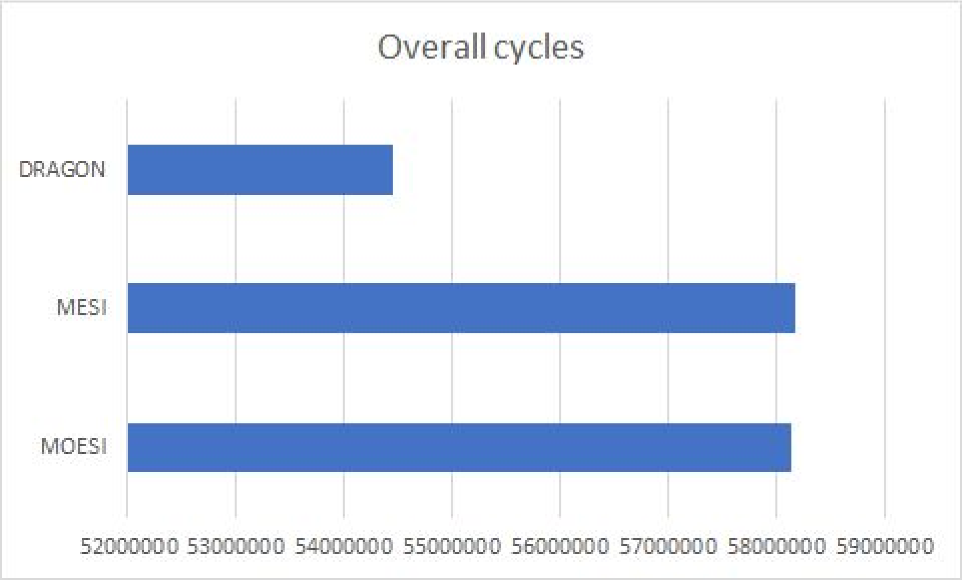
Optimizing DRAGON

We will do the same method for DRAGON:

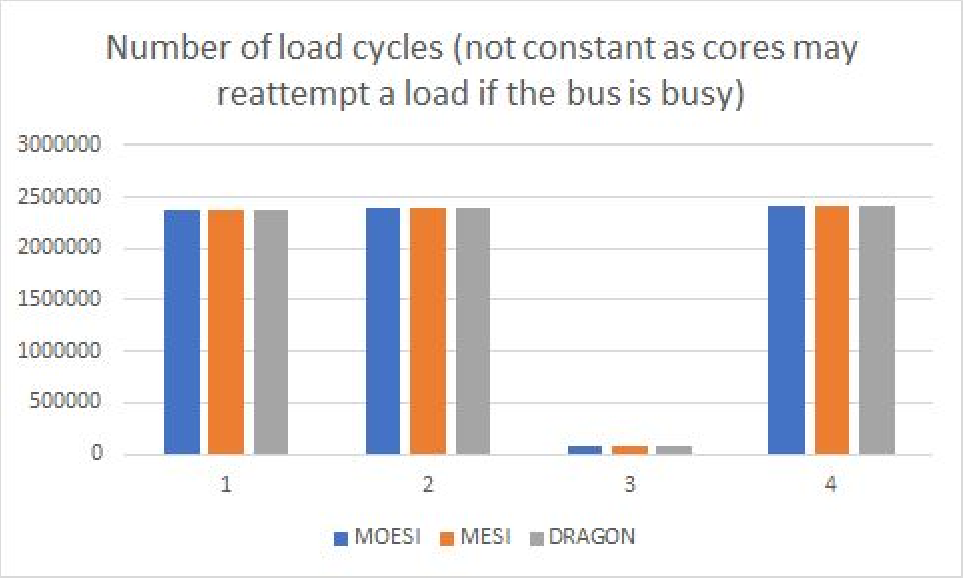
Block size / Associativity	16 bytes	32 bytes	64 bytes	128 bytes
1	93352997	79997897	70206698	
2			62638049	
8			56022077	
16			55437284	**54446364**
32				

We can see that the cache configuration giving the best result is here a 16-way set associative cache with 128 bytes block size. This is the configuration we will choose when comparing the protocols.

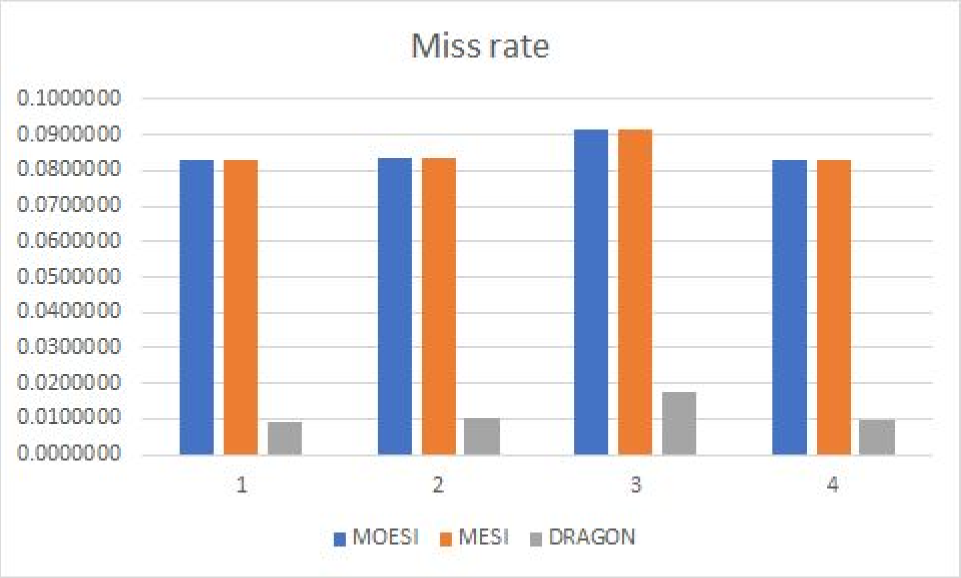
Comparing the three protocols



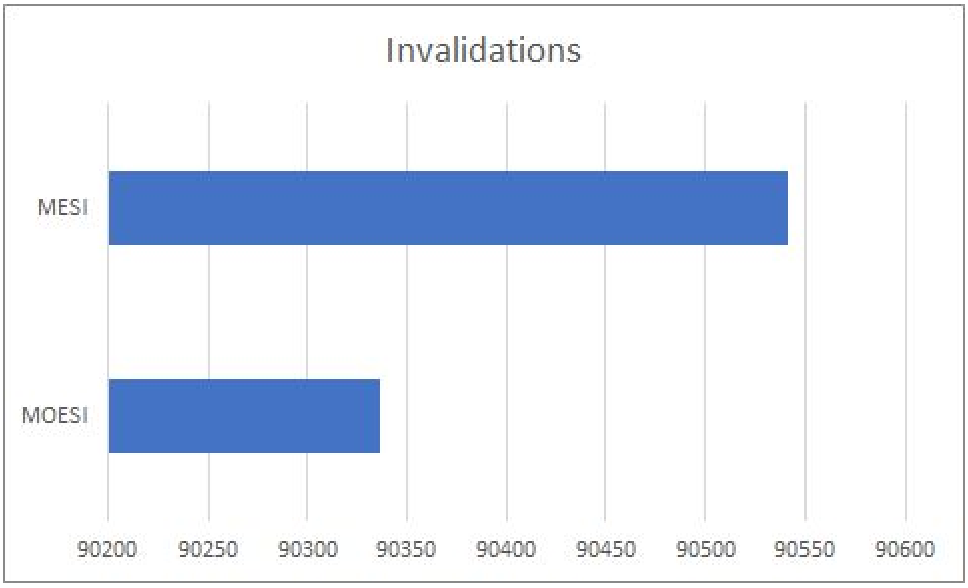
Again, we can first observe that the Dragon protocol is the one allowing the fastest execution. The second one is MOESI.

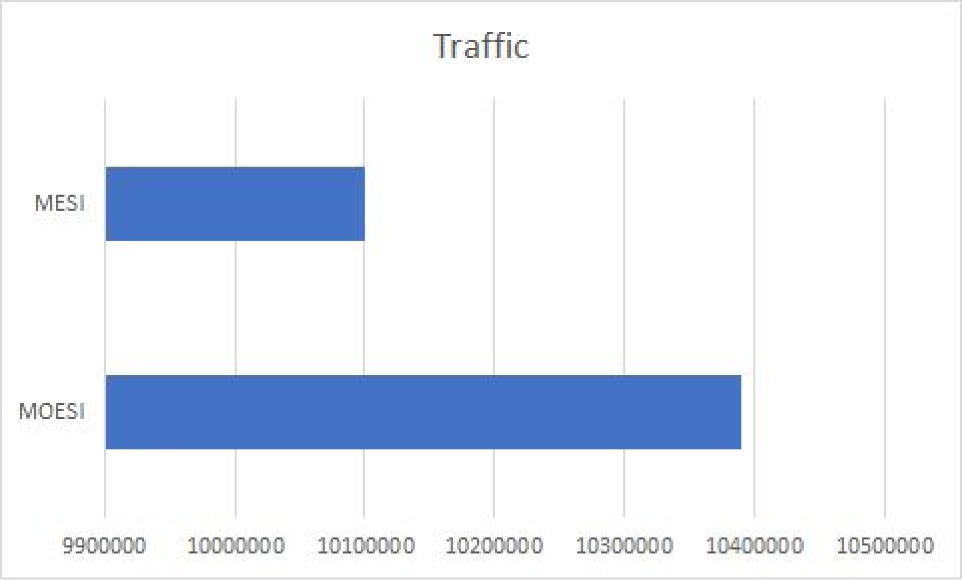


We can observe the great difference between core number 3 and other cores. We can perfectly explain this thanks to our previous static analysis: there are simply fewer memory instructions in core 3.



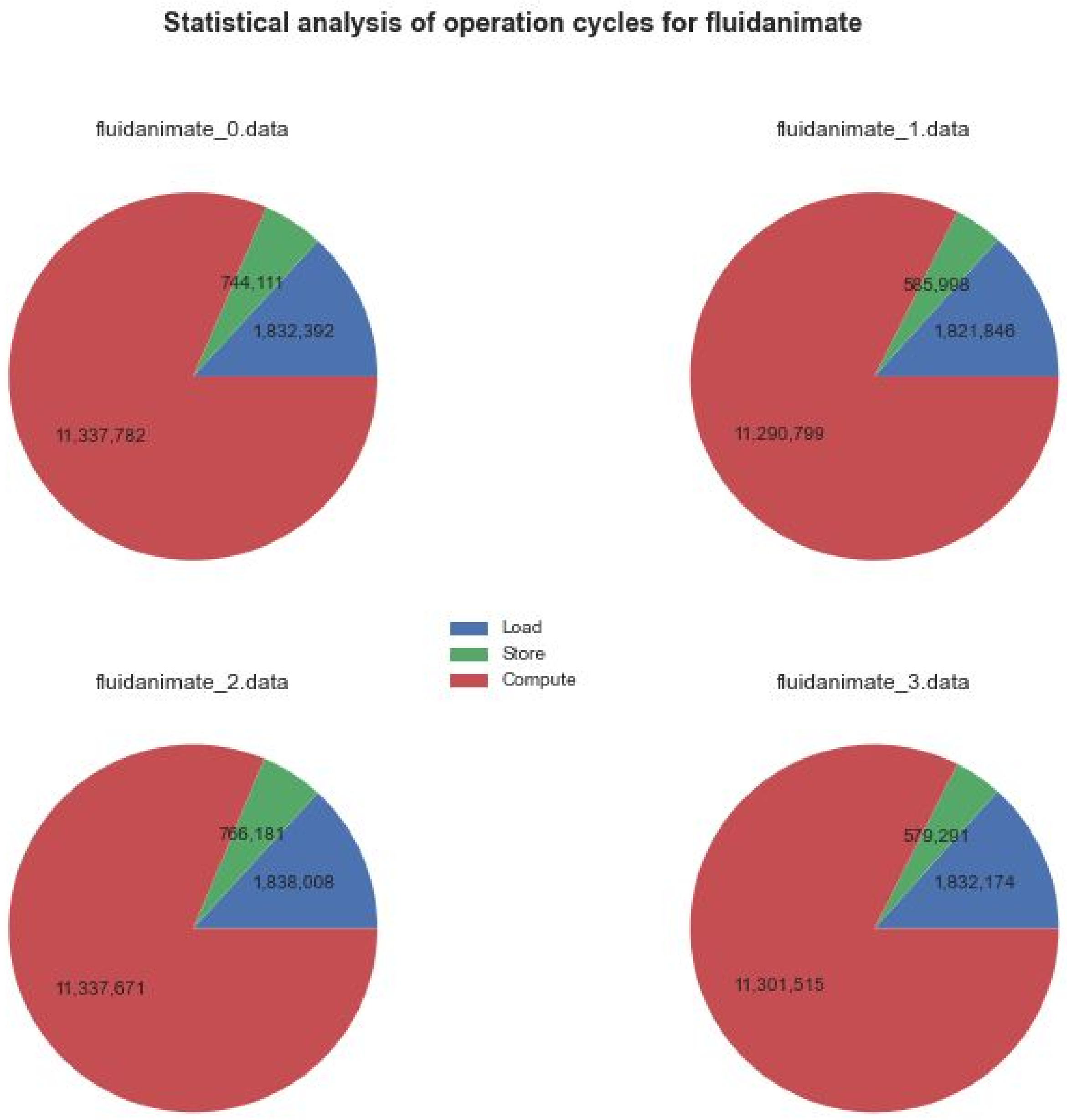
The miss-rate shows us a major advantage of the DRAGON protocol, again. We will see later that this situation is normal.





Again, the number of invalidations for MESI and MOESI is very close. However, we can observe significantly higher data traffic on the bus for the MOESI protocol. It is explainable as the MOESI has an additional cache to cache transfers.

Fluidanimate



We notice two facts :

- all the 4 processors have similar statistics
- compared to the 2 other benchmarks, this one has fewer loads

Optimizing MESI

In this part, we will try to find the best parameters for the MESI cache coherence protocol and this specific program. The cache parameters are the cache size, the associativity, and the block size. We will keep the cache size to 4kb and tweak the other parameters. We will aim to improve the overall execution cycle.

Block size / Associativity	16 bytes	32 bytes	64 bytes	128 bytes
1	135199851	117910874	84958504	69824591
2	128741379	107397293	73480115	61021422
8	127447655	104052496	67163476	52343008
16	127835200			**50473907**
32	128521436			

We can see that the cache configuration giving the best result is here a 16-way set associative cache with 128 bytes block size. This is the configuration we will choose when comparing the protocols.

It seems that enhancing the block size is not that beneficial. The spatial locality benefice may be limited to this benchmark.

Optimizing MOESI

We will do the same method for MOESI :

Block size / Associativity	16 bytes	32 bytes	64 bytes	128 bytes
1	135165405	117878708	84943169	71764980
2	128715408	107368020	73463814	61007051
8	127447545	104024938	67145110	52328268
16	127835733			**50470057**
32	128534110			

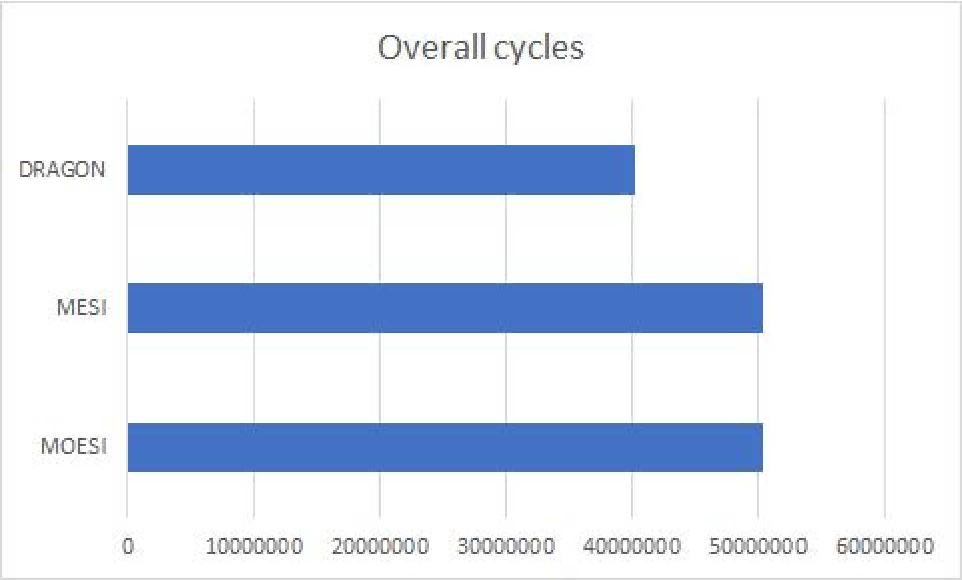
Optimizing DRAGON

We will do the same method for DRAGON:

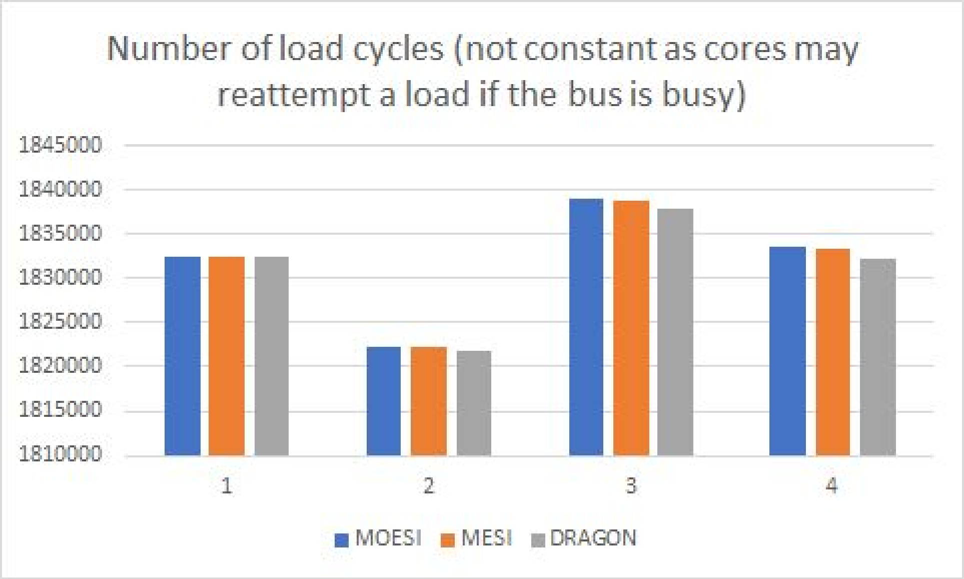
Block size / Associativity	16 bytes	32 bytes	64 bytes	128 bytes
1	96808312	83431036	64758547	
2			55421461	
8			50165947	
16			49731535	**40274164**
32				

We can see that the cache configuration giving the best result is here a 32-way set associative cache with 16 bytes block size. This is the configuration we will choose when comparing the protocols.

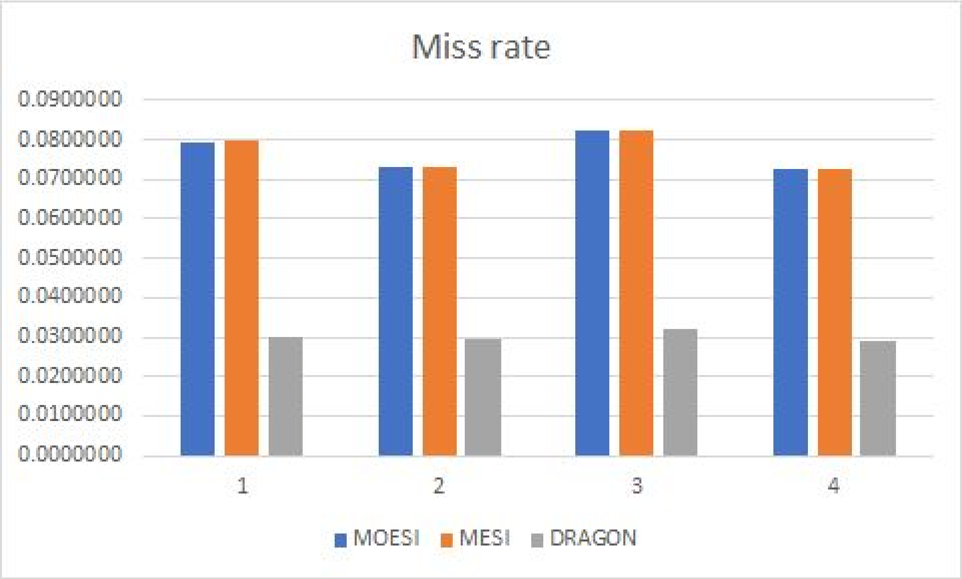
Comparing the three protocols



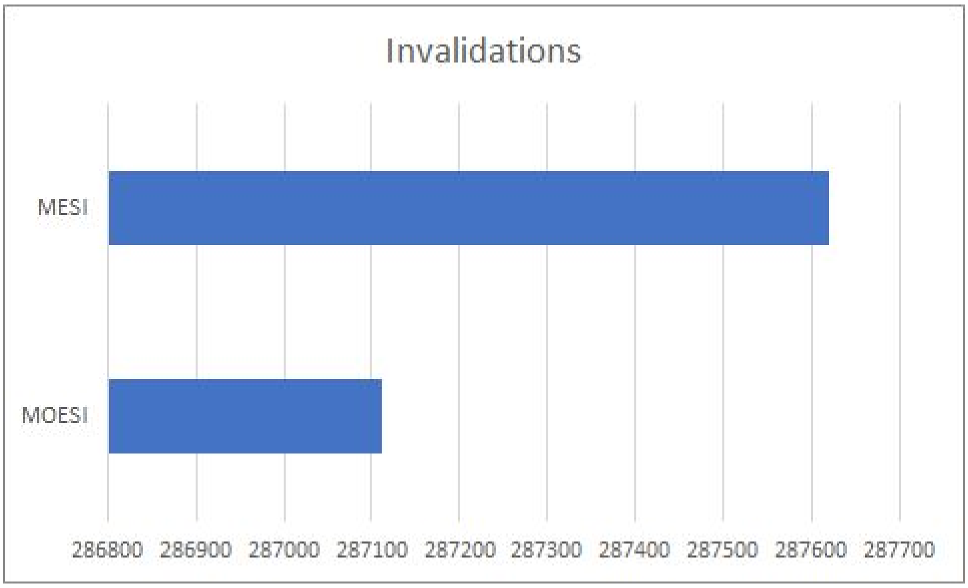
The same pattern repeats for this benchmark trace with Dragon protocol having a lower overall execution cycle.

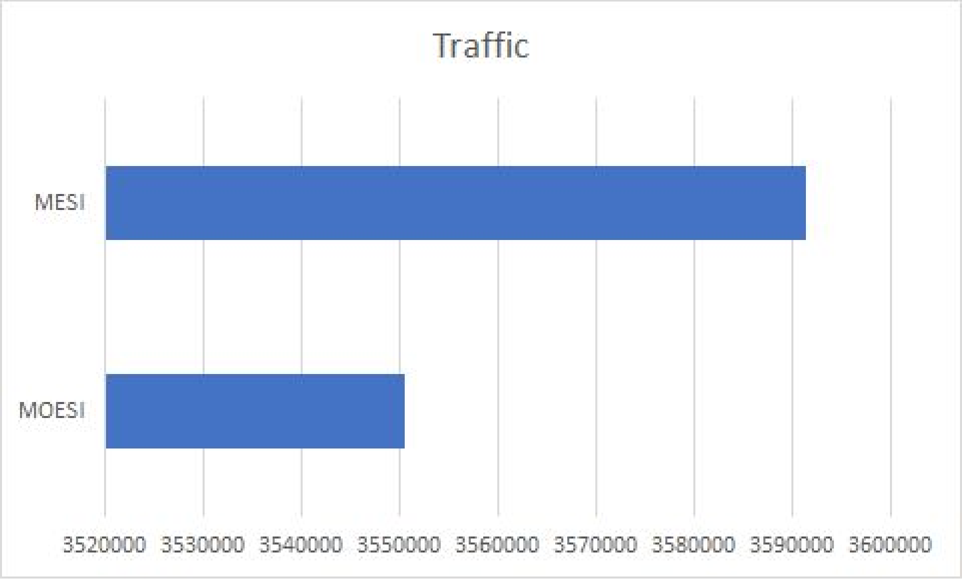


Once again, the uneven distribution of load cycles among the cores is evidence of unfair bus bandwidth allocation.



The miss rate for Dragon protocol is once again significantly lower than MESI protocol.





Again, the number of invalidations for MESI and MOESI is very close. However, we can observe higher data traffic on the bus for the MESI protocol.

Conclusion

MESI, being a write invalidate protocol, is more efficient when there are many subsequent writes to the same cache block after an initial write. For these writes to the same block, only one invalidation will be issued, no further bus transactions will be generated for other processors.

Dragon, being a write update protocol, is more efficient when there are many subsequent reads to the same cache block after a write. All other cached values are updated once the writes complete, hence other processors read the same block consecutively no bus transactions will be generated as they already have the values.

As a result, benchmark trace with more consecutive writes to the same cache block after an initial write would see a noticeable lower amount of bus transactions for MESI. Benchmark traces with more consecutive reads on the same cache block would thus produce fewer bus transactions for Dragon.

The miss rate is generally lower for Dragon as Dragon protocol has two processor transactions (prRdMiss and prWrMiss) to handle read and write misses. Also, due to the absence of the Invalid state, no block will be invalidated due to a write by a processor, hence indirectly lowers the chance of getting misses. With constant updates after write, all processors will be accessing up-to-date values further reducing the miss chance.

The number of updates is not generated for Dragon protocol due to a potential implementation flaw, that is an area of improvement for this project.