CS4223 Assignment 2: Design and Implementation of Tracing Simulator for MESI, MOESI and Dragon Cache Coherency Protocols

JULIUS PUTRA TANU SETIAJI (A0149787E), KYLE TIMOTHY NG CHU (A0155181J)

Contents

1 CHOICE OF PROGRAMMING LANGUAGE

Initially, we tried to implement the trace simulator using Python. However, after a while, we realised that it gets unwieldy, unreadable, and difficult to extend given the complexity of implementing a trace simulator. Particularly, while Python has enumerations, they are not as powerful as Algebraic Data Types (ADT) in other languages, especially with regards to associated values. Python's lack of pattern matching also makes implementing a finite state machine (FSM) a hassle. While Python's dynamic typing allows more rapid prototyping, a better compile-time typecheck would be really useful to prevent bugs.

Haskell seems to fit the bill really well here. However, since we are essentially simulating a mutable, eagerly-evaluating von Neumann architecture machine, Haskell's characteristic of being immutable and lazy can become an impediment. We need a language that has some of Haskell's features such as being strongly and statically typed, pattern matching and ADT, while still allowing us to have mutability. There are some candidates, such as Scala, Rust and OCaml. For these reasons as well as familiarity, completeness of the standard library and availabilities of third party libraries we chose to implement our tracing simulator using **Scala**.

2 IMPLEMENTATION

Our code can be divided into 2 large parts: common code for all the protocols, and the specific code for the trace simulator of each cache coherency protocols.

Overall, our code uses the delegation pattern, where each device can choose to become a delegate of another device. This way, devices can have a two-way communication of some sort. The devices themselves are designed as Finite State Machines (FSM) to allow for more robust code.

There are some potential issues that may prevent correctness in this concurrent system, namely deadlock, livelock, starvation and race conditions [4]. As we explain the design of the devices in our simulator, we will mention how we try to prevent these problems from happening.

2.1 `Main.scala` and `Protocol.scala`

This is the main driver code of the simulator. It takes in arguments passed by user and parses them into the appropriate types, such as `Int`, `String` and `Protocol`. `Protocol` is actually defined inside `Protocol.scala`, which is an enumeration of supported cache coherency protocols. In our case we have `MESI` and `Dragon`, as well as MOESI for the advanced task. If all the arguments are parsed successfully, the appropriate tracing simulator is then invoked.

2.1.1 `Debug.scala`. This file provides code for debug printing that can be elided for final production code.

2.1.2 `Address.scala`. This file provides the case class for addressing in an n-way associative cache through tag and set index.

2.2 `coherence.bus`

2.2.1 `Bus`. Across the different protocols, we use the same bus implementation. The design that we chose for our bus is **an atomic bus**, where one cache is granted exclusive access to the bus to perform one transaction (a transaction is defined as the cache granted exclusive access placing command on the bus, and then receiving response by other caches).

When a device wants to place command on the bus, it has to first request access from the bus, and the device has to be a `BusDelegate`. This ensures that a cache will have exclusive access to the bus, with the bus arbitrer acting like a mutex lock. The bus arbitrer uses a first come first serve arbitration policy to *prevent starvation*.

When the bus arbitrer decides that it is the turn for one `BusDelegate` to obtain exclusive access to the bus, the bus will invoke `busAccessGranted()` on the `BusDelegate` to obtain the command that the device wants to place on the bus. This is to *prevent race condition*, because the transition of the states of the cache lines are not actually atomic even though the bus is atomic. Consider the case in MESI protocol when Processors P1 and P2 write to cache line A simultaneously (both need to issue `BusUpgr` to move the line from the `S` state to the `M` state). Suppose P1 wins bus access and places `BusUpgr` on the bus. P2 is still waiting for bus access to place its own `BusUpgr`, but it cannot proceed because P1 has exclusive access to the bus. P2 then receives `BusUpgr` from `P1`, thus it must invalidate

line A according to the MESI protocol. As such, P2 must also change its pending `BusUpgr` request to a `BusRdX`. With our design, we will only ask the cache what command to place on the bus when they are granted access, thus allowing the caches to change the command that they send between the moments access is requested and access is granted.

After a device places a command on the bus, `onBusCompleteMessage` method on all other devices as `BusDelegate` will get invoked. This allows them to make internal state transitions in response to the command on the bus, as well as place their response on the bus. The bus will arbitrarily grant one device to place their response on the bus. Once the device having exclusive access to the bus decided that it has received enough responses, it will indicate to the bus that it wishes to relinquish its exclusive access. This fact is propagated to all other devices through invocation of the `onBusTransactionEnd` of the `BusDelegate`. The bus will then pick another pending access request, and perform the same cycle all over again. Note that to *avoid livelock*, the device that obtains exclusive access must be allowed to complete before exclusive access is relinquished, thus our design of letting the device decide when to relinquish exclusive access of the bus. Consider two processors P1 and P2 writing to cache block B using MESI cache coherency protocol. First, P1 acquired access to the bus and issues `BusRdX`, resulting in P2 invalidating its cache line. However, without exclusive ownership, this scenario can happen: before P1 can perform the write, P2 acquires bus, issues `BusRdX`, resulting in P1 invalidating its cache line, and so on so forth, which results in a livelock.

Note that in our bus design, each device connected to the bus is not blocked on requesting access to the bus. They are still able to service incoming transactions while waiting to be granted exclusive access to the bus. This is done to *prevent protocol-level deadlock*, sometimes called *fetch-deadlock* [4].

Our bus is implemented as an FSM with 4 states:

(1) `Ready` – no cache has exclusive access to the bus (the mutex lock is not acquired)
(2) `ProcessingRequest` – Processing a message from a cache, since only a finite amount of data can be transferred to the bus in a cycle
(3) `RequestSent` – A message has been posted on the bus, and the bus is accepting replies from the other devices
(4) `ProcessingReply` – Processing a reply from a cache, since only a finite amount of data can be transferred to the bus in a cycle

The state transition can be seen in Figure 1.

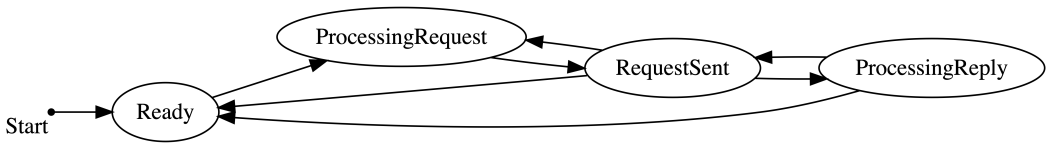


Fig. 1. State Transition for the Bus

2.2.2 `BusDelegate`. Specifies the hooks in a device that wants to hear back from the `Bus` regarding events on the Bus. Of interest is the `hasCopy` method, which is invoked in order

to check the "SHARED" OR line, required by the protocols to check whether a particular cache line already exists in the other caches.

2.2.3 `BusStatistics`. This is a "template" for each protocol to provide a class that takes note of statistics in the `Bus`

2.2.4 `MessageMetadata`. A case class that stores the message that a cache wants to place on the bus, as well as the address affected by that message. The size of each message if not explicitly given is assumed to be 1 byte long.

2.2.5 `ReplyMetadata`. A case class that stores the reply from other caches, as well as the size of the reply. The size here will be used in the calculation of how long it takes to transmit the reply and associated data on the Bus.

2.3 `coherence.cache`

This package contains classes necessary to build a cache with a Least Recently Used (LRU) cache eviction policy and where each cache line stores a state.

2.3.1 `LRUCache`. This is a simple cache with LRU cache eviction policy that can be used for each set in an n-way associative cache. It is implemented on top of Java's built-in `LinkedHashMap` which uses a hash table and linked list to keep track of the order of when an item was last used.

2.3.2 `CacheLine`. This is a simple case class representing a cache line. It is generic in the type parameter `State`, and stores only the state of this cache line.

2.4 `coherence.devices`

This package contains mostly the "template" for each type of specific devices. This helps to keep our code DRY[1] as the common code among all the protocol are put here.

2.4.1 `Device`. A simple "template" for all devices in the simulator. It only requires that every device has a method `cycle()` which is invoked to notify the device to advance one cycle and perform what they need to do in a cycle.

2.4.2 `Memory`. This is the "template" for a memory in each protocol. The memory will always return false as it is the identity value in the OR operation on the shared line, thus the memory will not affect the "SHARED" OR line.

2.4.3 `Processor`. This is a concrete class that represents a processor in the simulator. Each processor owns a cache, where the procesor is the cache's `CacheDelegate`. It is an FSM with 4 states, with transitions as shown in Figure 2:

- `Ready` – the processor is ready to execute the next instruction.
- `Operation` – the processor is currently executing a non-memory instruction
- `Cache` – the processor is waiting for the result of a memory instruction from its cache
- `Finished` – the processor has no more instruction to execute

[1] Don't Repeat Yourself, one important Software Engineering Principle

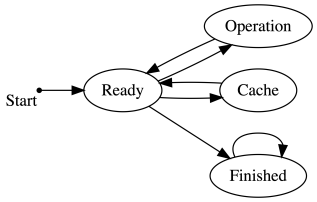


Fig. 2. The state diagram for the processor

2.4.4 `ProcessorOp`. This trait is effectively an ADT representing the three possible instructions: the memory instructions `Load` and `Store` which has the address as the associated value as well as the non-memory instruction `Other` with the number of cycles required for execution as the associated value. The associated singleton object of the `ProcessorOp` trait provides a function to parse each line in the input trace into a `ProcessorOp`.

2.4.5 `Cache`. This is the "template" for the protocol-specific implementation of the cache. The cache is an FSM with 8 states, whereby each state will inform the cache about what to do with regards to the bus:

- `Ready` – the cache is ready to accept commands from the processor
- `WaitingForBus` – the cache has requested access to the bus but is waiting for that access to be granted
- `WaitingForReplies` – the cache has exclusive access to the bus and has placed a message on the bus, and is waiting for replies from the other caches.
- `WaitingForWriteback` – the cache has exclusive access to the bus and has snooped a Flush request, and is currently waiting for the memory to acknowledge that the writeback succeeded.
- `WaitingForResult` – the cache has the cache line requested, and that cache line is valid, and the cache is currently experiencing its hit latency.
- `WaitingForBusPropagation` – the cache sent a message on the bus and is waiting for the message to reach the other caches.
- `EvictWaitingForBus` – the cache needs to evict a cache line, has requested access to the bus and is currently waiting for that access to be granted.
- `EvictWaitingForWriteback` – the cache currently has exclusive access to the bus and has placed the flush command on the bus and is waiting for the memory to acknowledge the writeback.

The state transitions are shown in Figure 3

2.4.6 `CacheDelegate`. This is the "template" for a cache delegate, which requires only one method, `requestCompleted` which is invoked when the operation is complete.

2.4.7 `CacheOp`. An ADT representing the possible cache operations.

2.4.8 `CacheStatistics`. This is a "template" for each protocol to provide a class that takes note of statistics in the `Cache`

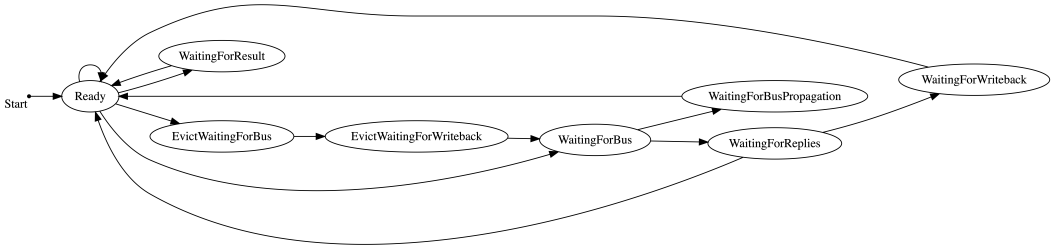


Fig. 3. The state diagram for the cache

2.5 MESI Protocol (`coherence.mesi`)

The MESI protocol consists of four states: Modified (M), Exclusive (E), Shared (S), and Invalid (I). The state transition diagram can be found in Figure 4. For any given pair of caches, the permitted states of a given cache line is given in Table 1

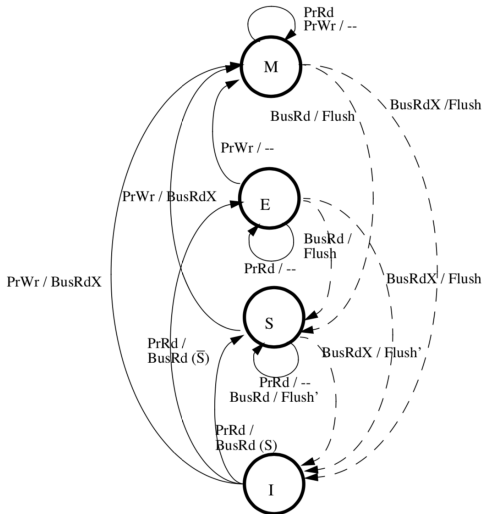


Fig. 4. State Transition Diagram of the MESI Cache Coherency Protocol

Table 1. Permitted States for a Pair of Caches in MESI Protocol

	M	**E**	**S**	**I**
M	No	No	No	Yes
E	No	No	No	Yes
S	No	No	Yes	Yes
I	Yes	Yes	Yes	Yes

2.5.1 Message. There are 4 possible messages to be placed by a cache that gained exclusive access to the bus:

(1) `BusRd` – a read request to a cache block
(2) `BusRdX` – a read exclusive request to a cache block, typically when a cache wants to write to a cache block it does not already have.
(3) `BusUpgr` – a request to gain read exclusive access to a shared cache block that a cache already has.
(4) `Flush` – a request to write back the cache block to the main memory

2.5.2 Reply. There are 4 possible replies to be placed by a cache or memory to a Message:

(1) `Flush` – when a cache block must be written back to the main memory
(2) `FlushOpt` – when a cache block is posted on the bus to supply it to another cache
(3) `MemoryRead` – when the memory posts a cache block on the bus
(4) `WritebackOk` – acknowledgement from the memory that a cache block has been written successfully

2.5.3 Memory. The memory in our MESI Protocol simulator snoops on the bus and will do one of these three things:

(1) Do nothing if the command is `BusUpgr`
(2) Read from the address in the command placed by a cache if the command is `BusRd` or `BusRdX`
(3) Write back to memory if the command is `Flush`

If the memory detects that another cache is providing the cache line instead, e.g. if the state is S, then the memory will cancel the current read request. If the memory detects that a cache replied with OK, it will cancel the read request, and start the writeback process instead.

2.5.4 Cache. This is the specific implementation of abstract class `Cache` from `coherence.devices` for the MESI protocol according to the state transition diagram in Figure 4

In our implementation, in case of cache line that is shared, the one supplying the block will be arbitrarily picked, depending on the one that was able to place its reply on the bus first. When other caches detect that the block content has been posted on the bus, they will change state so as to not double-send the block.

2.6 MOESI Protocol (`coherence.moesi`): Optimisation to Basic MESI Protocol

Some processor architectures such as UltraSPARC [7] and AMD Opteron [6] use an optimised version of the MESI protocol by adding one more state: O for Owned, representing a cache line that is both modified and shared. The idea is to allow cache having a modified cache line to supply that data to another cache without performing a slow writeback to memory. The state transition diagram can be found in Figure 5. For any given pair of caches, the permitted states of a given cache line is given in Table 2.

2.6.1 Differences Compared to MESI. We use the MESI code as a base, and then applying the necessary patches to it. Firstly, we needed to add one more state `O`. We can reuse all the `Message` and `Reply` from MESI, as what differs is the behaviour in response to those

messages and replies. In total, we changed about 17 lines to create the relevant behaviour change required to turn the MESI protocol into the MOESI protocol.

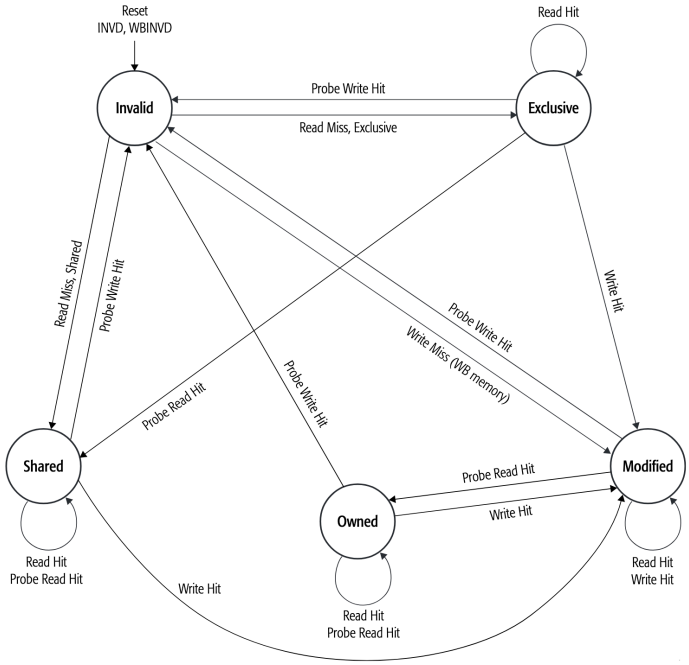


Fig. 5. State Transition Diagram of the MOESI Cache Coherency Protocol

Table 2. Permitted States for a Pair of Caches in MESI Protocol

	M	**O**	**E**	**S**	**I**
M	No	No	No	No	Yes
O	No	No	No	Yes	Yes
E	No	No	No	No	Yes
S	No	Yes	No	Yes	Yes
I	Yes	Yes	Yes	Yes	Yes

2.7 Dragon Protocol

The Dragon Protocol is a write-update based cache coherence protocol that was first introduced along with the Dragon Multi-processor [2]. Unlike write-invalidate protocols like MESI, Dragon updates shared cache lines at each write instead of just invalidating them.

2.7.1 Message. There are 3 possible messages to be placed by a cache that gained exclusive access to the bus:

(1) `BusRd` – a read request to a cache block

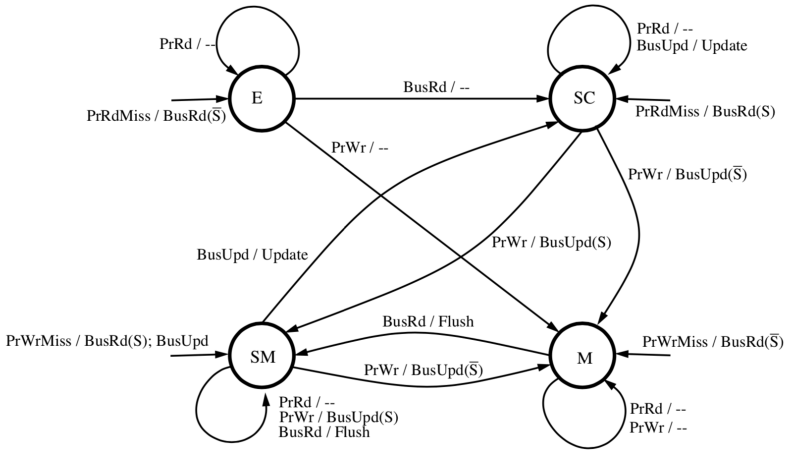


Fig. 6. State Diagram for the Dragon Protocol

Table 3. Permitted States for a Pair of Caches in Dragon Protocol

	E	**Sc**	**Sm**	**M**
E	No	No	No	No
Sc	No	Yes	Yes	No
Sm	No	Yes	No	No
M	No	No	No	No

(2) `BusUpt` – a request to caches that contain the same cache block to update their local copies
(3) `Flush` – a request to write back the cache block to the main memory

2.7.2 Reply. There are 4 possible replies to be placed by a cache or memory to a Message:
(1) `Flush` – when a cache block must be written back to the main memory
(2) `MemoryRead` – when the memory posts a cache block on the bus
(3) `WritebackOk` – acknowledgement from the memory that a cache block has been written successfully

2.7.3 Memory. The memory in our Dragon Protocol simulator snoops on the bus and will do one of these three things:
(1) Do nothing if the command is `BusUpt`
(2) Read from the address in the command placed by a cache if the command is `BusRd`
(3) Write back to memory if the command is `Flush`

3 QUANTITATIVE ANALYSIS

3.1 Workload Analysis

Before analysing the results of our simulations, it is important to first briefly analyse the nature of the workloads we used in these experiments. The workloads used are from the PARSEC benchmark suite which is used for benchmarking the performance of multi-core processors [3]. The table below provides a summary of the key characteristics of the benchmarks.

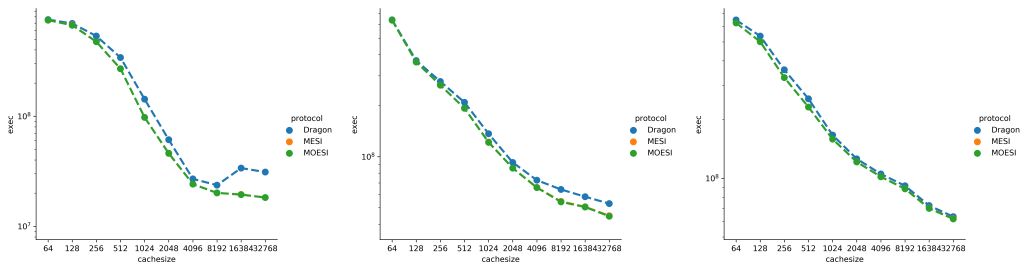
Table 4. Key characteristics of the benchmarks [3]

Benchmark	Granularity	Data Sharing	Data Exchange
blackscholes	Coarse	low	low
bodytrack	Medium	high	medium
fluidanimate	Fine	low	medium

- *blackscholes* - This benchmark consists of a main thread which spawns several worker threads that act on independent sections of data. As a result the majority of shared memory among the threads is between the main thread and the worker threads. Since the worker threads process data independently of each other, there is little communication between the threads.
- *bodytrack* - This benchmark has multiple threads working on the same set of data, specifically it uses a thread pool parallelism model.
- *fluidanimate* - This benchmark has multiple threads working on the same set of data. Due to the nature of the workload, there is a significant amount of inter-thread communication which scales with the number of threads [3]

3.2 Experimental Results & Analysis

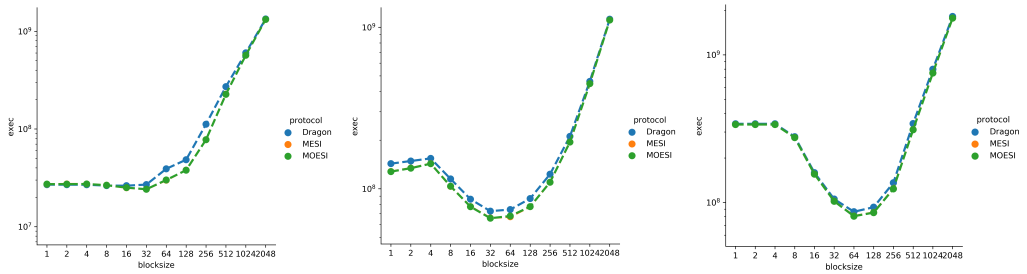
We ran our simulations with varying cache sizes, block sizes and set associativity. The results are presented and discussed in the sections below:



(a) blackscholes (b) bodytrack (c) fluidanimate

Fig. 7. Plot of total execution cycles against cache size with 2-way associative cache with 32 bytes block size, Note that the line for the MESI protocols may be difficult to see as it almost completely overlaps with the MOESI protocol.

3.2.1 Cache size. As seen in Figure 7, the MOESI/MESI protocol outperforms the Dragon protocol in all 3 benchmarks regardless of cache size. We believe that the reason for this is due to the unnecessary write updates on cache lines that are not read again after they were first cached. This is especially apparent in the *blackscholes* benchmark there is little shared memory. This problem is exacerbated as the cache size increases due to more of such entries remaining in the cache for longer periods of time before they are evicted. We can observe this effect in the *blackscholes* benchmark where the penalty of these unnecessary updates is dominant over the benefit gained from a larger cache size past 8kB. This causes the number of cycles taken to execute the application to increase after that point. The *fluidanimate* shows much more promising results for the Dragon protocol since there is a high amount of inter-thread communication in the workload. However, we observe that the MOESI/MESI protocol still performs slightly better. One reason for this could potentially attributed to multiple sequential write instructions being executed before a read takes place which is also another source of unnecessary write updates.

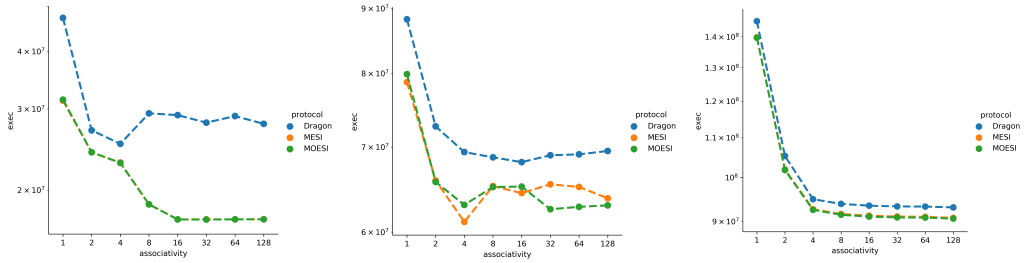


(a) blackscholes (b) bodytrack (c) fluidanimate

Fig. 8. Plot of total execution cycles against block size with 2-way associative cache with 4 kilobytes cache size

3.2.2 Block size. As seen in Figure 8, the MOESI/MESI protocol once again out performs the Dragon protocol in all 3 benchmarks regardless of block size which can be attributed to the same reasons stated above. One interesting phenomenon that is observed here is how performance sharply drops past a certain block size in all 3 benchmarks. In the MOESI/MESI protocol, this is most probably due to the increased number of invalidation misses as an invalidation causes the entire cache block to be invalidated which could have contained data that would be needed in the next few instructions. In the Dragon protocol, this can be attributed to a higher overhead associated with updating the cache blocks for each write update.

3.2.3 Associativity. As seen in Figure 9, the MOESI/MESI protocol once again out performs the Dragon protocol in all 3 benchmarks regardless of block size which can be attributed to the same reasons stated above. We can also see that the performance gained from increasing associativity flattens out in all 3 benchmarks for all the protocols. This is most likely due to the fact that the sets are not being fully utilized, thus there is diminishing returns from increasing the associativity.



(a) blackscholes (b) bodytrack (c) fluidanimate

Fig. 9. Plot of total execution cycles against associativity with 32 bytes block size with 4 kilobytes cache size

Table 5. Amount of data traffic in bytes on the bus in MESI and MOESI protocols as associativity increases in the *bodytrack* benchmark

Associativity	MESI	MOESI
1	21,421,632	21,420,672
2	18,001,696	18,000,992
4	17,203,296	17,199,904
8	16,968,896	16,967,200
16	16,978,272	16,975,296
32	17,053,344	17,057,184
64	17,152,480	17,057,184
128	17,205,280	17,202,208

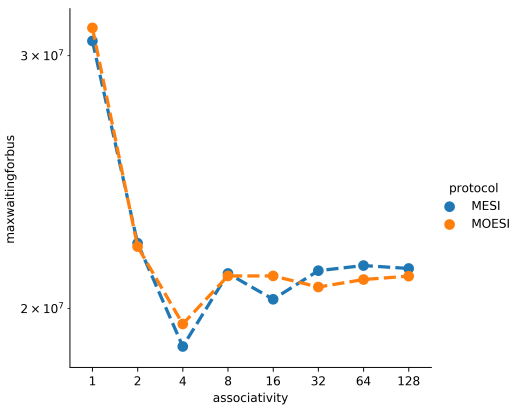


Fig. 10. Plot of number of cycles spent waiting for access to bus by the processor with the highest number of execution cycles against associativity with 32 bytes block size and 4 kilobytes cache size in the *bodytrack* benchmark.

More interestingly, we can see that the MESI protocol is able to outperform the MOESI protocol in the *bodytrack* benchmark noticeably when associativity is 1, 4 and 16. We try

to first investigate this matter by looking at the amount of traffic in the bus (see Table 5), but MOESI in general decreases the amount of data traffic on the bus in the *bodytrack* benchmark. Our hypothesis is that MOESI's worse performance is caused by the increased contention in the bus. To check whether our hypothesis is correct, we log the number of cycles spent waiting for access to the bus by the processor with the highest number of execution cycles (as this processor is the bottleneck). From the results, we can indeed see the correlation of the time spent waiting for access to the bus in Figure 10 and the total execution time in Figure 9b Thus, although MOESI decreases the amount of traffic on the bus, the bus traffic is more concentrated at some points in time, increasing the bus contention.

3.3 Concluding statements

From the experiments conducted, we have shown that the MESI protocol and its variants are able to outperform the Dragon protocol across all the given benchmarks. While there are most certainly benchmarks that favour the Dragon protocol that were not tested here, these benchmarks would need to exhibit specific properties such as an interleaving of reads and writes across many different processors which may not be applicable to more general applications. We can also see that in general, MOESI performs slightly better than MESI. There are specific cases as explained in Subsubsection 3.2.3 where MESI can outperform MOESI, but this does not happen too often.

These reasons are possibly why most modern architectures make use of MESI and optimised protocols derived from it such as MOESI and MESIF rather than update-based cache coherency protocols. For example, AMD's AMD64 processors use MOESI [1], ARM Cortex-A57 uses MESI in its L1 cache and MOESI in its L2 cache [5], while Intel uses MESIF in its 64-bit processors [8].

REFERENCES

[1] AMD. 2006. AMD64 architecture programmer's manual volume 2: System programming.

[2] Russell R. Atkinson, Edward M. McCreight, and Edward M. McCreight. 1987. The Dragon Processor. *SIGARCH Comput. Archit. News* 15, 5 (Oct. 1987), 65–69. https://doi.org/10.1145/36177.36185

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[4] David Culler, Jaswinder Pal Singh, and Anoop Gupta. 1998. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[5] ARM Holdings. 2014. ARM Cortex-A53 MPCore Processor, Technical Reference Manual.

[6] Bruce Jacob, Spencer Ng, and David Wang. 2007. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[7] Sun Microelectronics. 1997. UltraSPARC™ User's Manual. *STP1030-UG, Sun Microelectronics (January 1996)* (1997).

[8] Michael E Thomadakis. 2011. The architecture of the Nehalem processor and Nehalem-EP SMP platforms. *Resource* 3, 2 (2011), 30–32.