CS4223 Assignment 2

A Haskell-based Multicore Cache Coherence Simulator:

Alvian Prasetya and Sriram Sami

Introduction **2**
- Inputs 2
- Outputs 2
  - Simulation-Global 2
  - Processor/Cache-Specific 2
  - Bus-specific 2

Simulator Programming Language **3**

Implementation **4**
- High-Level Component Overview 4
- Component Description 5
  - Main 5
  - SimulatorCore 5
  - Processor 5
  - Protocol 5
  - Bus 5
  - Cache 5
- Method Call Structure 6
- MESI and Dragon Protocol Implementation 7
  - MESI Load State Machine 7
  - MESI Store State Machine 9
  - Dragon Load State Machine 11
  - Dragon Store State Machine 13
- Advanced Task: Illinois MESI 15
  - MESI + Cache-to-cache (Illinois) Load State Machine 15
  - MESI + Cache-to-cache (Illinois) Store State Machine 16
- Data Structures 17
  - Cache and Cache Block representations 17
  - Bus Transaction Queue representation 17
- Implementation Difficulties 18

Quantitative Analysis **20**
- Pre-benchmark 20
- Benchmarks Results and Analysis 21

Conclusion **33**

Appendix I: Sample Raw Data Dump (Associativity = 1 across Benchmarks) 34

Introduction

This project aimed to produce a **trace-based cache coherence simulator** for **multi-core architectures** that aids in the analysis of cache coherence protocol choices and finding optimal design points for metrics like cache and block size.

Inputs

Protocol: MESI or Dragon
Trace Files: 4 trace files, one for each of the 4 processor cores
Cache Size: Entire L1 cache size in bytes
Associativity: Magnitude of cache associativity
Block Size: Size of a cache block in bytes

Outputs

Simulation-Global

Overall Execution Cycles:
Total number of cycles taken for entire trace to complete

Processor/Cache-Specific

Compute cycles: Total number of cycles executing OtherInstructions
Number of load/store instructions: Entire L1 cache size in bytes
Number of idle cycles: Cycles spent executing cache operations
Cache miss rate: Overall % miss rate

Bus-specific

Data traffic on bus: Total amount of data sent across the bus in bytes
Number of invalidations/updated: Total BusRdX or Bus Update requests
Number of private data accesses: Access count to own internal cache blocks
Number of public data accesses: Access count to blocks "owned" by other caches

Simulator Programming Language

Haskell was used as the core language for our simulator. Haskell is known for its very strong and expressive type system, extreme focus on function purity, referential transparency, functional style, object immutability and default use of lazy evaluation. This forces good software engineering practices since techniques like unsafe typecasting (to deal with bad class design), using potentially uncaught errors to propagate information and holding multiple mutable references to an object so that its state can be changed at any point in the program are not possible, so the program must be designed well from the start. Additionally, a good deal of information is encodable in the type system so that incorrect states simply will not compile. For example:

```
data MESIState = MESIWaitCacheRead | MESIWaitCacheWrite | MESIWaitBusTr |
MESIWaitMemoryRead | MESIWaitMemoryWrite | MESIWaitCacheRewrite | MESIDone

data DragonState = DragonWaitCacheRead | DragonWaitCacheWrite |
DragonWaitBusTr | DragonWaitMemoryRead | DragonWaitMemoryWrite |
DragonWaitCacheRewrite | DragonDone

data ProtocolStates = MESIProtocol MESIState | DragonProtocol DragonState
```

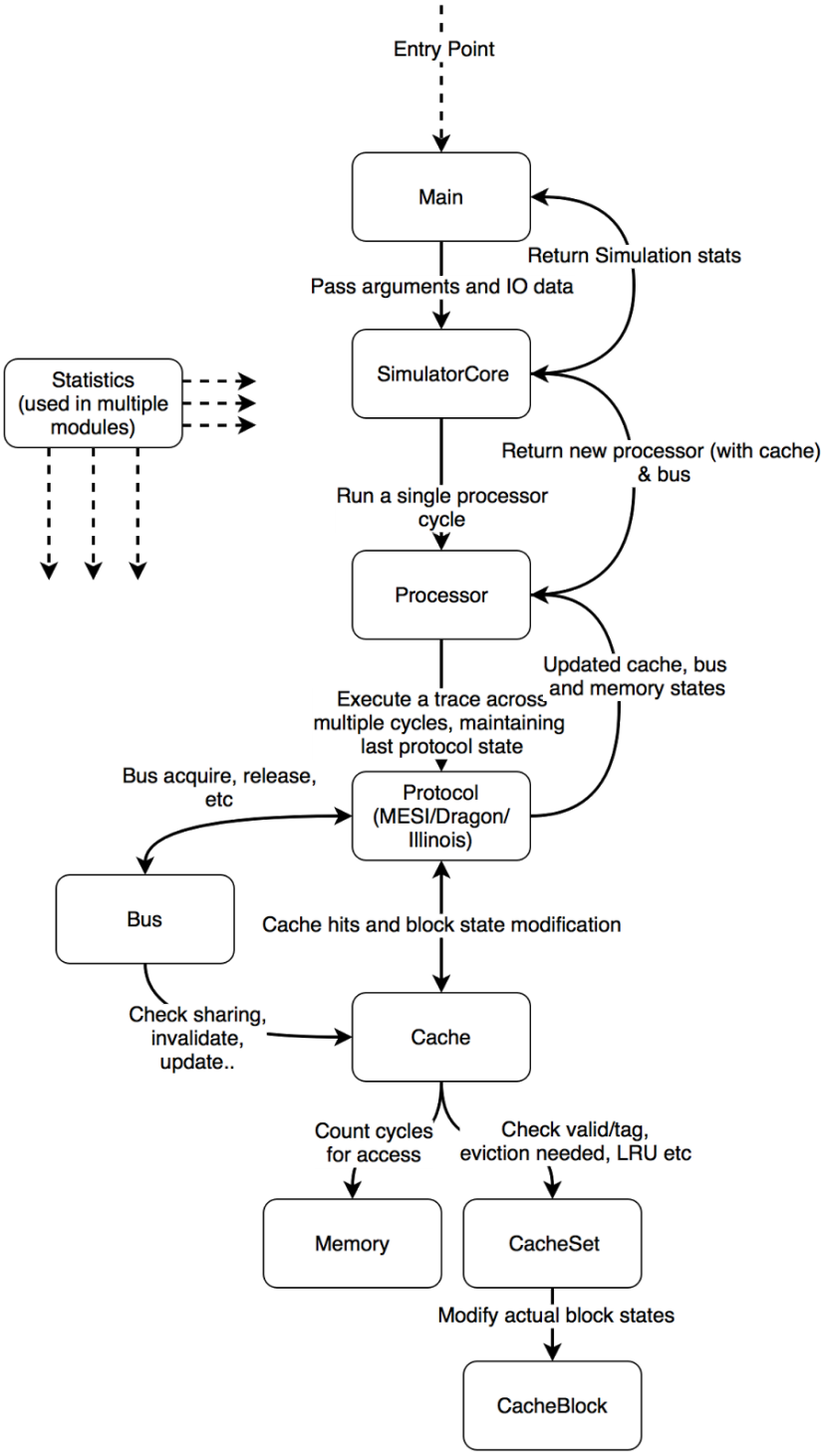
Haskell uses **algebraic data types** to encode information in its type system. In this example, the **`ProtocolStates`** type is used in our **`Processor`** class to keep track of which protocol-level state the execution of the current trace is in. The definitions of **`ProtocolStates, DragonState`** and **`MESIState`** ensure that the Dragon and MESI states can be accepted together for generic functions that operate on the protocol state, but both types and all of their sub-states must be handled somewhere within the function (a.k.a we must write code for processing all types), and this avoids errors where not all cases are considered.

A final example of this is **the absence of a *null* type.** Therefore, any *null reference errors* are not possible. An example of this is the function signature: **`busGetBlockState :: MemoryAddress -> Cache -> (Maybe BlockState)`**, which is a function that takes in a Memory Address (type-constrained to an Int32 as well!), a cache, and returns a **`(Maybe BlockState).`** Since getting a block state can fail when there is no such block for that address inside the cache, instead of return a **null** object that might cause errors, Haskell forces us to use a **`Maybe`** type that has one of two values in this case: **`Just (BlockState)`** or **`Nothing.`** The Just type indicates a successful return, whereas Nothing indicates failure. *Both cases must be explicitly handled for the program to compile,* avoiding dangling null values.

Both the challenges and advantages of Haskell drew us to attempt this project using the language to see if it would be a good choice for simulators.

Implementation

High-Level Component Overview



Module overview for the simulator - Haskell allows encapsulation of data types and associated functions in modules but there is no internal mutable state.

Component Description

Main

Receives data from the outside world (command line arguments, trace file names etc) and passes it to **SimulatorCore**

SimulatorCore

SimulatorCore first completes running all IO actions like reading the trace files, then passes all arguments in pure form to the **startSimulatorPure** part of the module. This is the IO/pure barrier and from this point on all functions are pure. Core now runs each individual cycle for the simulator, and calls each processor in turn during a cycle.

Processor

Processor executes a single cycle when called by Core. It either runs the OtherInstruction trace by itself, waiting for the appropriate number of cycles as Core calls it every cycle, or it receives a Load/Store instruction and passes it with the current state machine state to the correct Protocol layer module and method: `(MESIProtocol/DragonProtocol/IllinoisProtocol).(load/store)`. Protocol returns a new protocol state (based on each individual protocol's state machine) and a new cache, memory and bus instance (since objects in Haskell are immutable). Processor stores those that are in its own state and returns a modified instance of itself to Core, along with the current state of the bus.

Protocol

Protocol is the most complex component in the simulator, consisting of three possible instances: `MESIProtocol, DragonProtocol, IllinoisProtocol.` Its job is to receive the current trace being processed and the previous state machine state and appropriately call the Cache, Bus and its own instance of Memory to decide how to transition to further states. It returns a new cache, memory, bus, and the current state in its state machine to Processor.

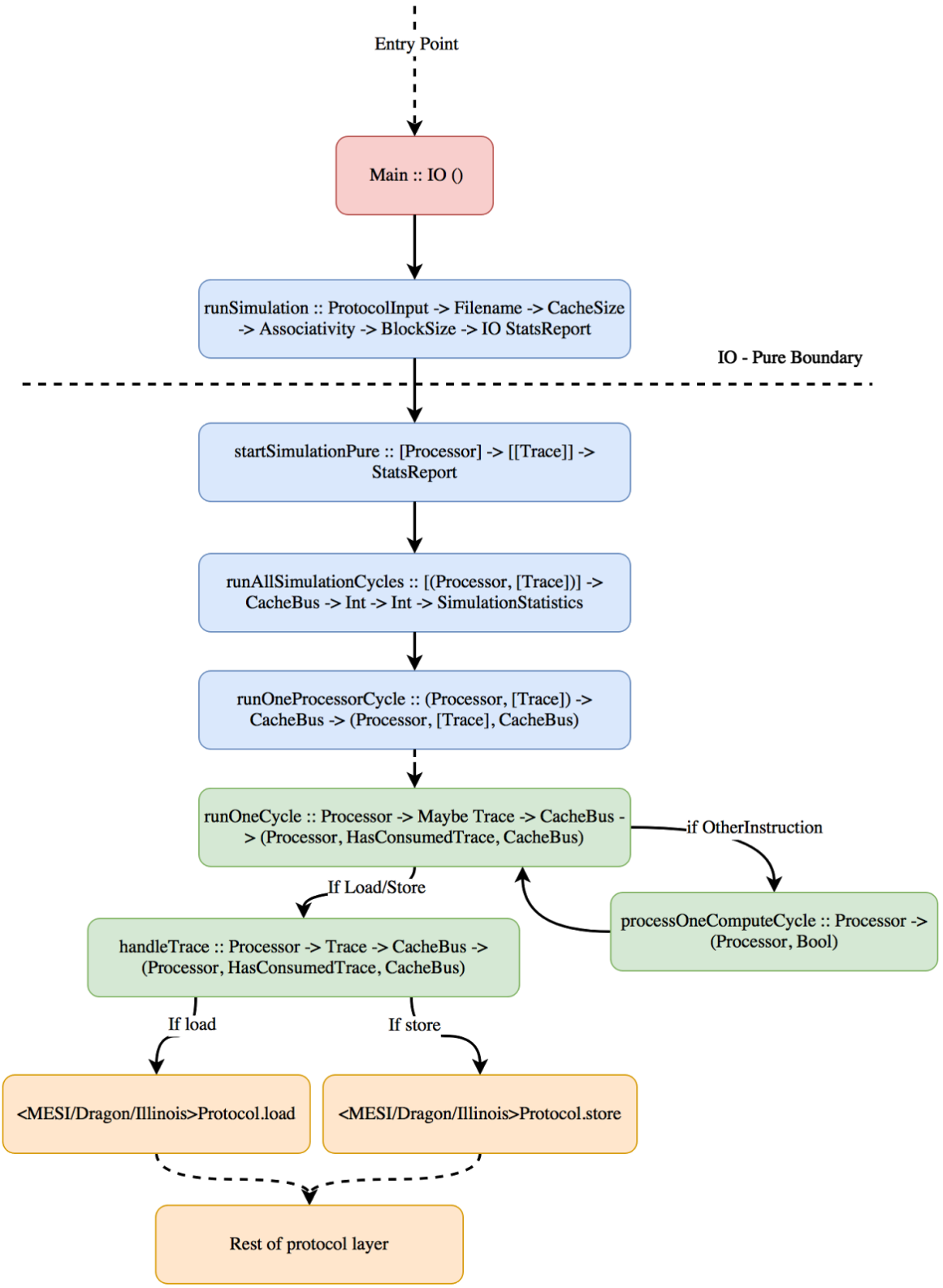
Bus

Bus mainly provides the *acquire, issue* and *release* methods to the Protocol layer so that each cache can lock down the bus, issue a transaction to the bus to request for sharing status, start invalidation, etc, and then release the bus to other caches to use through the Protocol layer.

Cache

The cache subsystem tracks block/set states, LRU operations, etc, and provides methods to the bus to conduct evictions/allocations/block state setting.

Method Call Structure

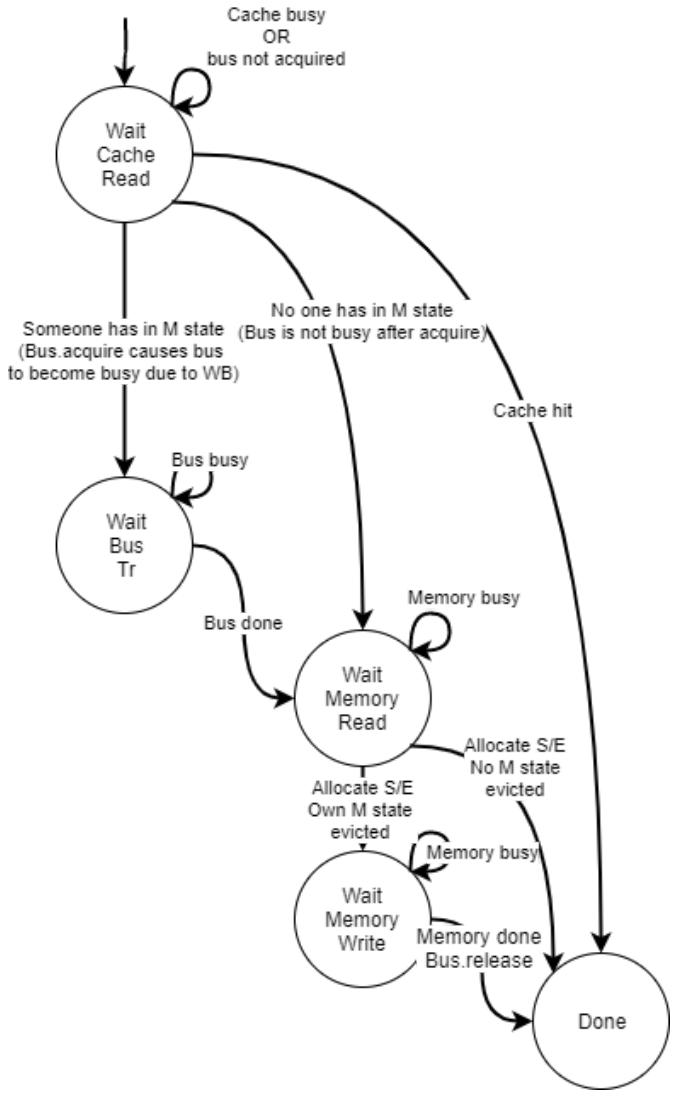


Legend: Red - IO, Blue - SimCore, Green - Processor, Orange - Protocol

Expanding on the previous component descriptions, this is an overview of the methods calls until we reach the protocol layer.

MESI and Dragon Protocol Implementation

MESI Load State Machine

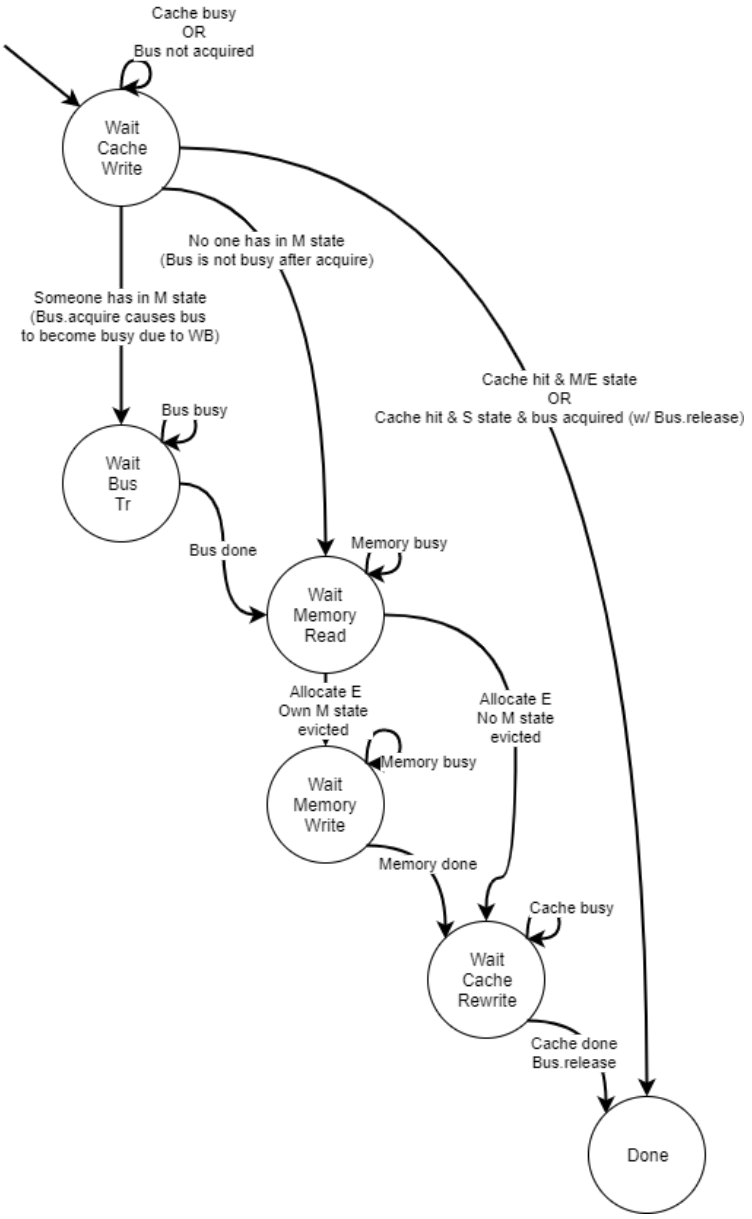


State Legend:

- **WaitCacheRead**: any load operation is always initiated by issuing a cache read and transitioning to the `WaitCacheRead` state. On this state, the protocol will keep checking the cache for read completion, upon which it may then transition to 3 possible states: `Done` (if the read was a hit), `WaitMemoryRead` (if the read was a miss and no one has a copy in M state, thus not requiring the bus to do a write-back), or `WaitBusTr` (if the read was a miss and another cache has a copy in M state, causing the bus to do write-back and be busy until WB is finished). Furthermore, a transition to `WaitMemoryRead` or `WaitBusTr` state requires the bus to be acquired successfully, otherwise it will keep trying to acquire and stay in `WaitCacheRead` state.

- **WaitBusTr**: this state is reached whenever the bus is busy after BusRd was issued (due to someone having a copy in M state, thus the bus needs to write-back this to memory). On this state, the protocol will keep checking the bus for busy status, when it is no longer busy it will issue a memory read to fetch the requested data and transition to `WaitMemoryRead` state.
- **WaitMemoryRead**: this state is always reached whenever a read miss occurs, as the protocol would need to fetch the data from memory to the cache. On this state, the protocol will keep checking the memory read for completion, upon which it will transition onto 2 possible states: `Done` (if allocation to cache was done without eviction of an M block), or `WaitMemoryWrite` (if allocation to cache was met with an eviction of M block from the same cache set, requiring a write-back to memory).
- **WaitMemoryWrite**: this state will only be reached whenever an allocation from memory was met with eviction of a block in M state, requiring a write-back to the memory and going to this state to wait for memory to finish. On this state, the memory will be checked for write completion, after which it will transition onto the `Done` state.
- **Done**: the ultimate state in the end of any load operation, when this state is reached, Processor will report back the load results along with all updated states to SimulatorCore.

MESI Store State Machine

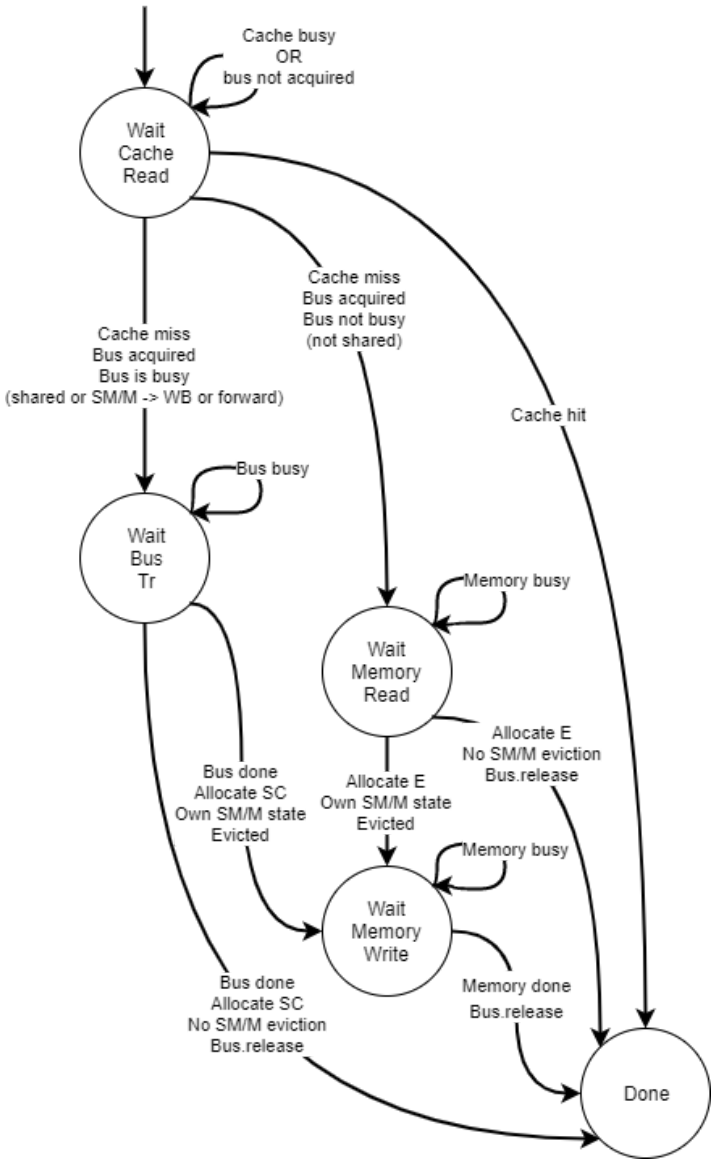


State Legend:

- **WaitCacheWrite**: any store operation is always initiated by issuing a cache write and waiting for it in `WaitCacheWrite` state. On this state, the protocol will keep checking for cache write completion, upon which it will transition to 3 possible states: `Done` (upon cache hit), `WaitMemoryRead` (upon cache miss and no one having a copy in M state, thus not requiring the bus to do a write-back), or `WaitBusTr` (upon cache miss and another cache having a copy in M state, requiring the bus to write-back and going to busy state). Furthermore, transitions to `Done` state on cache hit S state, `WaitMemoryRead`, and `WaitBusTr` requires the bus to be acquired successfully, otherwise it will keep trying to acquire in `WaitCacheWrite` state.

- **WaitBusTr**: this state is reached whenever a the bus is busy after BusRdX was issued (due to someone having a copy in M state, thus the bus needs to write-back this to memory). On this state, the protocol will keep checking the bus for busy status, when it is no longer busy it will issue a memory read and transition to `WaitMemoryRead` state.
- **WaitMemoryRead**: this state is reached whenever a write miss occurs, since the data needs to be fetched from memory to the cache. On this state, the protocol will keep checking for memory read completion, upon which it will transition to 2 possible states depending on whether an M block was evicted during allocation. If an M block was evicted during allocation, it will issue a memory write to write-back the evicted M block and transition to `WaitMemoryRead`, otherwise it will issue a cache rewrite and transition to `WaitCacheRewrite` state.
- **WaitMemoryWrite**: this state can only be reached when an M block was evicted during allocation, requiring a write-back to the memory. On this state, the protocol will keep checking for memory write completion, upon which it will issue a cache rewrite and transition to `WaitCacheRewrite` state.
- **WaitCacheRewrite**: this state will finally be reached whenever a write miss occurs, since the data has just been allocated to the cache and requires a rewrite. On this state, the protocol will keep checking for cache rewrite completion, upon which it will transition to the `Done` state.
- **Done**: the ultimate state in the end of any store operation, when this state is reached, Processor will report back the store results along with all updated states to SimulatorCore.

Dragon Load State Machine



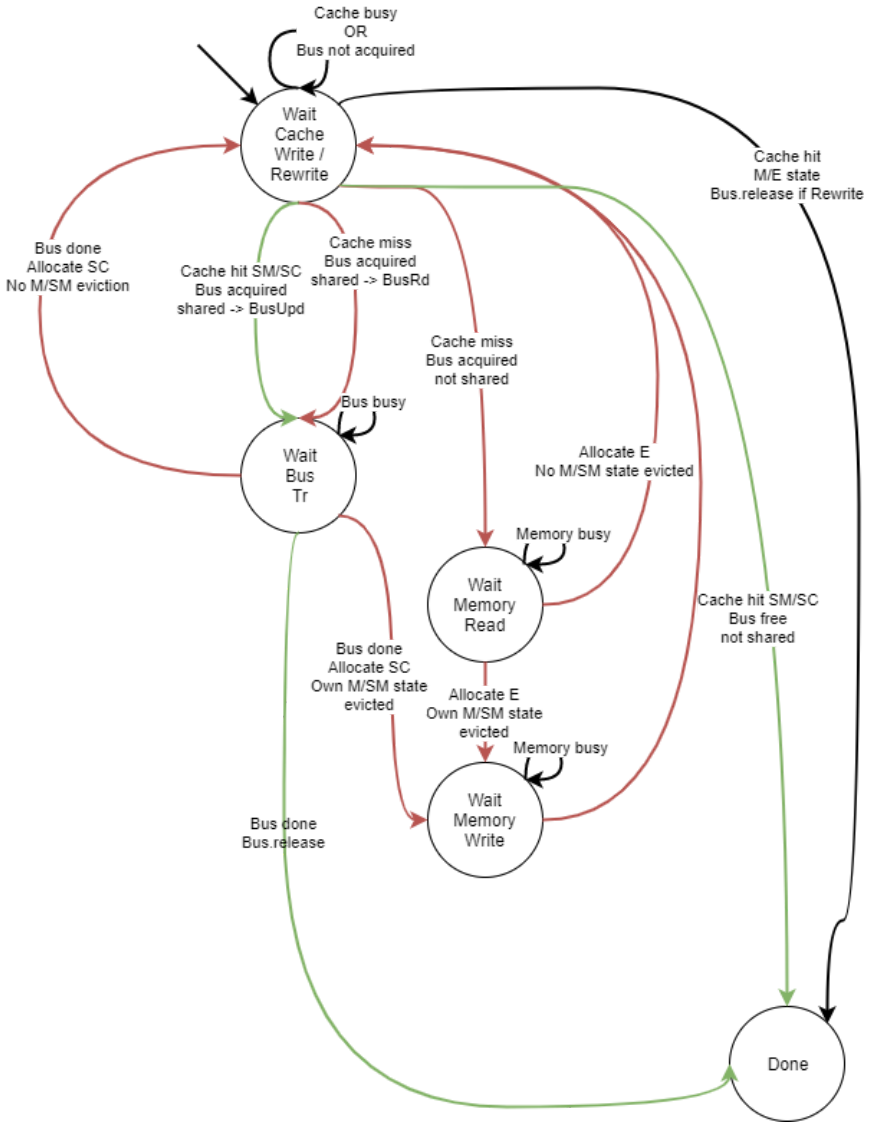
State Legend:

- **WaitCacheRead**: any load operation is always initiated by issuing a cache read and transitioning to the `WaitCacheRead` state. On this state, the protocol will keep checking the cache for read completion, upon which it may then transition to 3 possible states: `Done` (if the read was a hit), `WaitMemoryRead` (if the read was a miss and no one has a copy in M state, thus not requiring the bus to do a write-back), or `WaitBusTr` (if the read was a miss and another cache has a copy in M state, causing the bus to do write-back and be busy until WB is finished). Furthermore, transition to `WaitMemoryRead` or `WaitBusTr` state requires the bus to be acquired successfully, otherwise it will keep trying to acquire and stay in `WaitCacheRead` state.
- **WaitBusTr**: this state will be reached whenever the bus is busy after BusRd is issued (at least one copy exists in other caches, requiring the

bus to handle forwarding of data or write-back in case it is in M state). On this state, the protocol will keep checking for bus busy status, when it is no longer busy, it will transition to 2 possible states: `WaitMemoryWrite` (if an M block was evicted during allocation of shared data, requiring a write-back to be done) or `Done` (if no M block was evicted during allocation of shared data).

- **WaitMemoryRead**: this state will be reached whenever a read miss occurs and no other caches have a copy of the required data. On this state, the protocol will keep checking for memory read completion, upon which it will transition to 2 possible states: `WaitMemoryWrite` (if an M block was evicted during allocation of shared data, requiring a write-back) or `Done` (if no M block was evicted during allocation of fetched data).
- **WaitMemoryWrite**: this state will only be reached whenever an M block was evicted during allocation of shared data, requiring a write-back. On this state, the protocol will keep checking for memory write completion, upon which it will transition to `Done` state.
- **Done**: the ultimate state in the end of any load operation, when this state is reached, Processor will report back the load results along with all updated states to SimulatorCore.

Dragon Store State Machine



State Legend:

Note: Red/green paths are continuous (i.e. if you follow a red path from the start, it needs to be followed until Wait Cache Write/Rewrite/Done)

- **WaitCacheWrite**: any store operation is always initiated by issuing a cache write and waiting for it in `WaitCacheWrite` state. On this state, the protocol will keep checking for cache write completion, upon which it will transition to 3 possible states: `Done` (upon cache hit on M/E state or cache hit on SM/SC state with no other caches having a copy), `WaitMemoryRead` (upon cache miss and no one having a copy, necessitating a memory read), or `WaitBusTr` (upon cache miss and another cache having a copy, requiring the bus to write-back or cache hit on SM/SC state with other caches having a copy, requiring an update operation through the bus). Furthermore, transitions to `Done` state on cache hit SM/SC state, `WaitMemoryRead`, and `WaitBusTr` requires the bus to be acquired

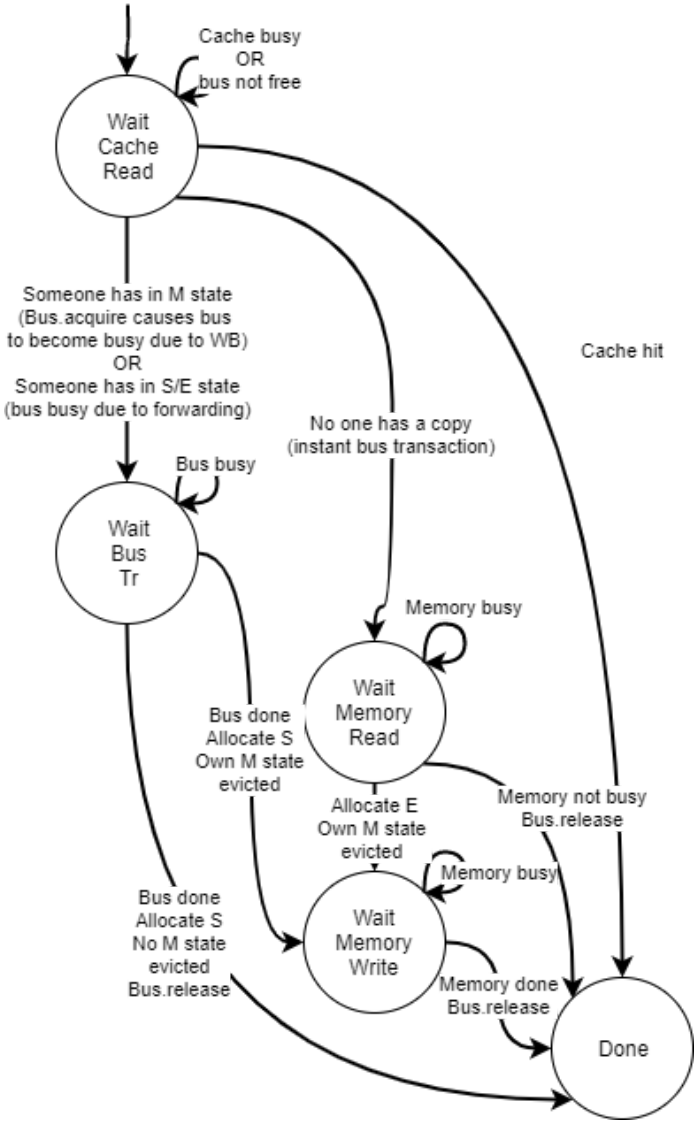
successfully, otherwise it will keep trying to acquire in `WaitCacheWrite` state.

- **WaitBusTr**: this state will be reached whenever a bus transaction (BusRd/BusUpd) is issued and the bus goes to busy state (due to shared copies, need to forward/update). On this state, the protocol will keep checking the bus busy status. When it is no longer busy, it will transition to 3 possible states: `Done` (if no allocation required because it was a write hit in SM/SC state), `WaitMemoryWrite` (if allocation causes eviction of M state block, requiring a write-back), or `WaitCacheRewrite` (if no M state block is evicted during allocation).
- **WaitMemoryRead**: this state will be reached whenever a write miss occurs and no other caches have a copy of it, requiring a memory read to be issued. On this state, the protocol will keep checking for memory read completion, upon which it will transition to 2 possible states: `WaitMemoryWrite` (if allocation causes eviction of M state block, requiring a write-back) or `WaitCacheRewrite` (if no M state block is evicted during allocation).
- **WaitMemoryWrite**: this state could only be reached whenever an M state cache block was evicted during allocation, requiring a write-back. On this state, the protocol will keep checking for memory write completion, upon which it will transition to `WaitCacheRewrite` to rewrite onto the previously allocated block.
- **WaitCacheRewrite**: this state will finally be reached whenever a write miss occurs, since the data has just been allocated to the cache and requires a rewrite. On this state, the protocol will keep checking for cache rewrite completion, upon which it will transition to the `Done` state.
- **Done**: the ultimate state in the end of any store operation, when this state is reached, Processor will report back the store results along with all updated states to SimulatorCore.

Advanced Task: Illinois MESI

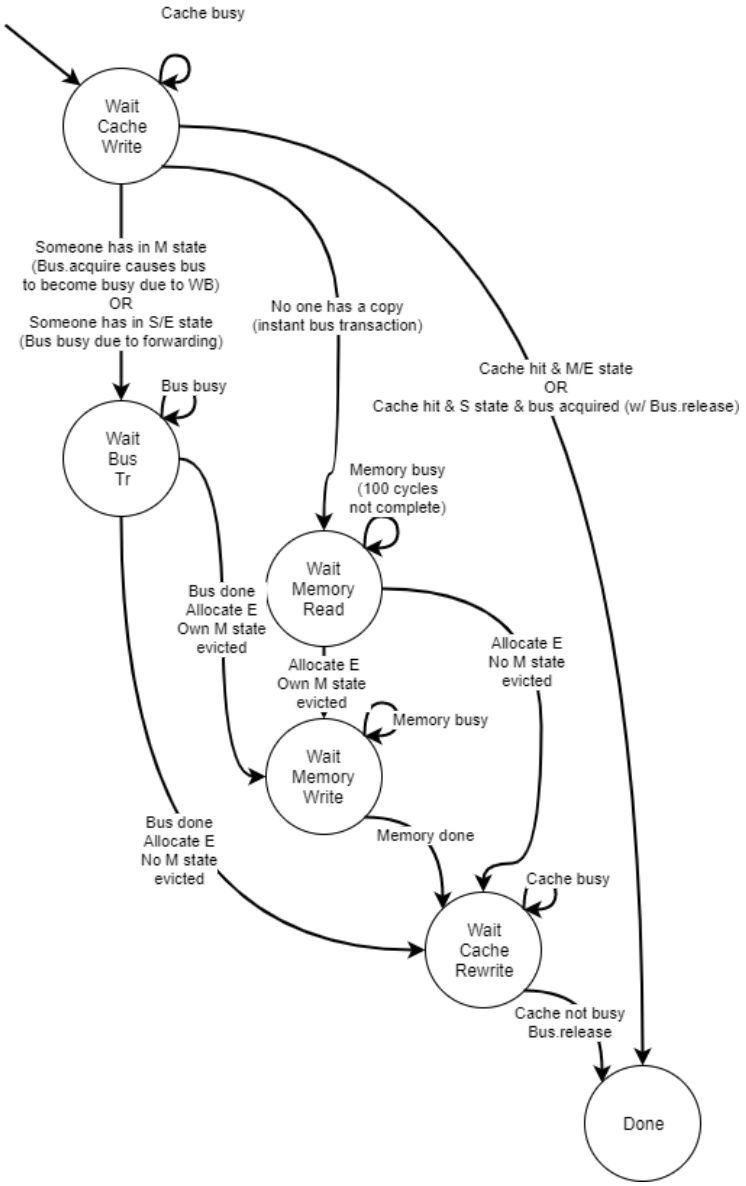
We conducted a literature search after noticing the Illinois MESI improvement mentioned during our lectures. The added benefit of Illinois MESI is that a cache-to-cache transfer will occur between a cache that has just encountered a miss and another cache if the other cache holds the relevant block; this removes the need for a costly memory access to receive this data. Therefore, we implemented Illinois MESI to observe the differences between these protocols.

MESI + Cache-to-cache (Illinois) Load State Machine



All the states are the same as in MESI Load State Machine, the only difference is whenever a cache read miss occurs and other caches happen to have a copy of the data, a cache-to-cache transfer will occur through the bus, eliminating the necessity to read from memory. Transitions to and from other states are exactly the same.

MESI + Cache-to-cache (Illinois) Store State Machine

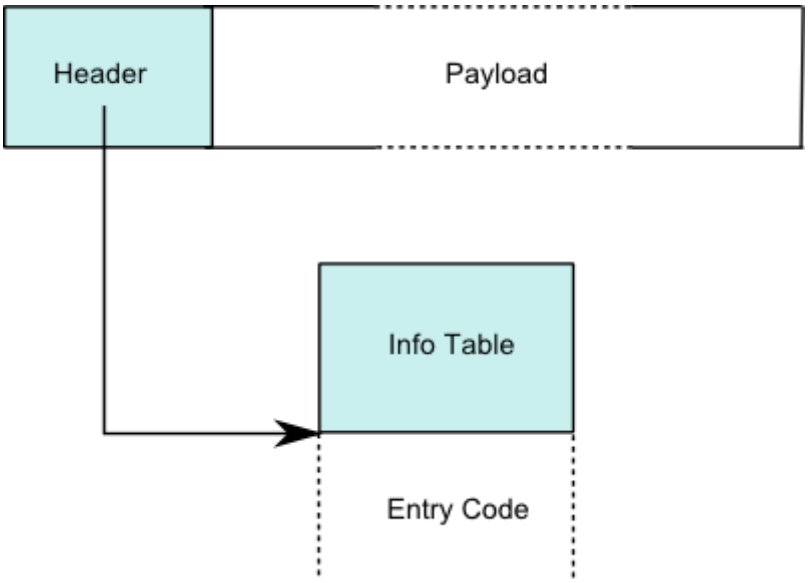


All the states are the same as in MESI Store State Machine, the only difference is whenever a cache write miss occurs and other caches happen to have a copy of the data, a cache-to-cache transfer will occur through the bus, eliminating the necessity to read from memory. Transitions to and from other states are exactly the same.

Data Structures

Cache and Cache Block representations

We initially used Haskell's **`Data.Array`** package to represent our **`Cache`**'s internal list of **`CacheSets`**, and our **`CacheBlock`**'s internal storage of memory addresses. However, we realized that every time we changed a **`CacheBlock`** state, the the entire **`Cache`** was reconstructed. This caused massive problems with garbage collection times far outstripping actual processing time. The image below shows the level of indirection required for heap-allocated objects.



We then shifted to usage of Haskell's **`Data.Array.Unboxed`** type. This removes one level of pointer indirection in each of our cache element representations and removes the need for the GC to scan through the array during its generational phases, but is only available for machine datatypes like Int32, Char etc. Fortunately we selected Int32 as our memory address representation. This alleviated some memory problems and helped with improving runtime.

Bus Transaction Queue representation

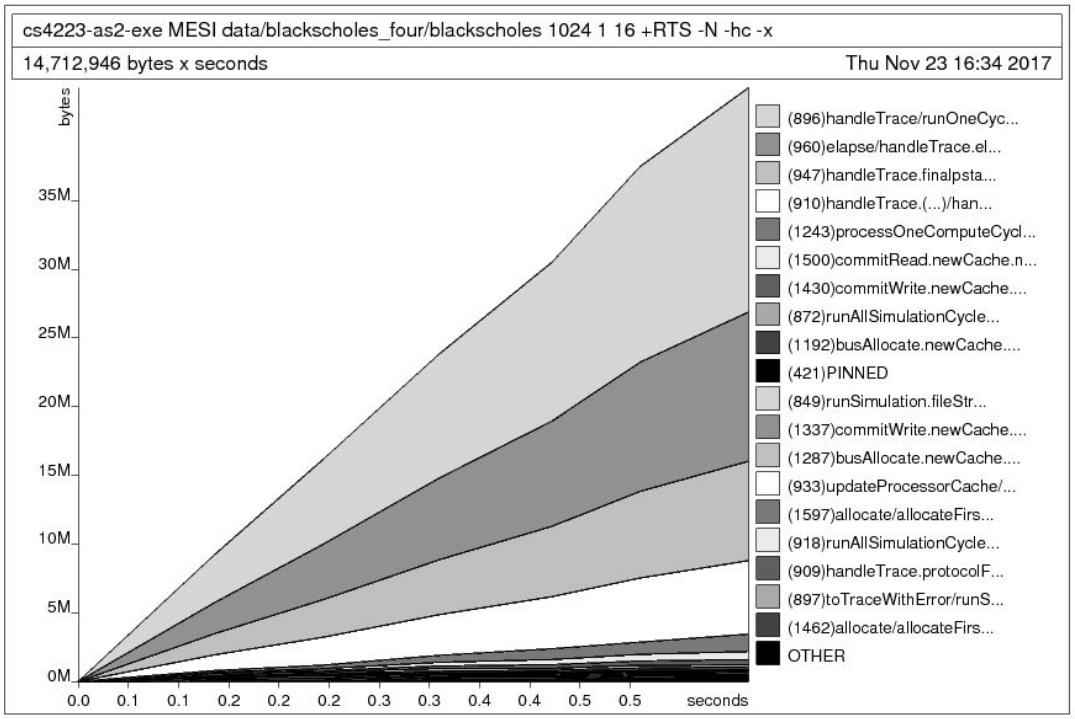
We used `Data.Sequence.Seq` from the `containers-0.5.10.2` package to represent the queue that stores pending bus transactions so that all bus transactions are served in the **first-come-first-served** order. This is internally represented as a *2-3 finger-tree annotated with sizes* which gives us O(1) enqueue, dequeue and peeking at both ends. This was fast and served its purpose.

Implementation Difficulties

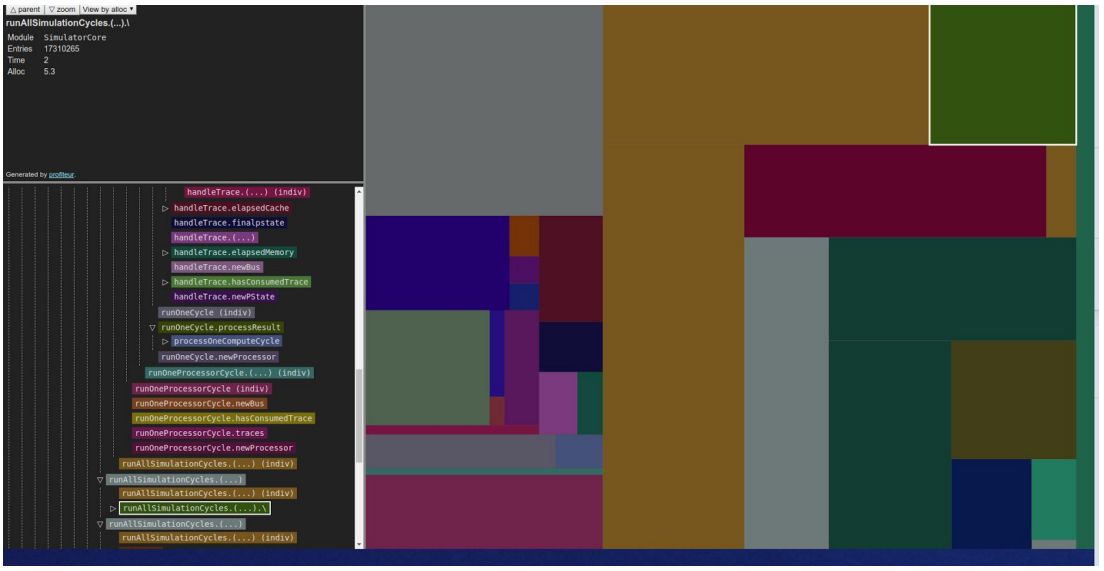
Due to our relative inexperience with Haskell, certain problems were beyond us to solve in this timeframe. The most significant problem was **time and space inefficiencies.**

In the latest Haskell surveys and reports, one of the most complained-about features of Haskell was the inability to reason about its time and space complexity easily. A continuing theme of our tests was massive heap space usage (in the order of **10s of GB**) that we could not conclusively find a solution to. Possibilities range from **excessive lazy evaluation** that causes a large build up of **thunks** (i.e. values that remain uncomputed for a long time), to **the GC struggling to re-create our Cache structure on every cycle** due to the copying required in pure Haskell, to a myriad of other ways that heap space can be mismanaged in Haskell due to the high level of abstraction it makes the programmer operate at.

We conducted extensive profiling with the Glasgow Haskell Compiler (GHC)'s own inbuilt tools and external utilities - including heap tracing:



And temporal analysis of the main cost-centers in the program:



However, all allocations looked normal - memory was being allocated at the points that were expected and time was spent in functions that were called most, also without surprise. The difficulty was in why the garbage collector could not release all the memory that became unused, somehow causing a buildup. We were not able to do more than alleviate the problem slightly, and therefore made some compromises during data analysis. However, the data analysis will still ultimately accurately depict the differences across cache coherence protocols.

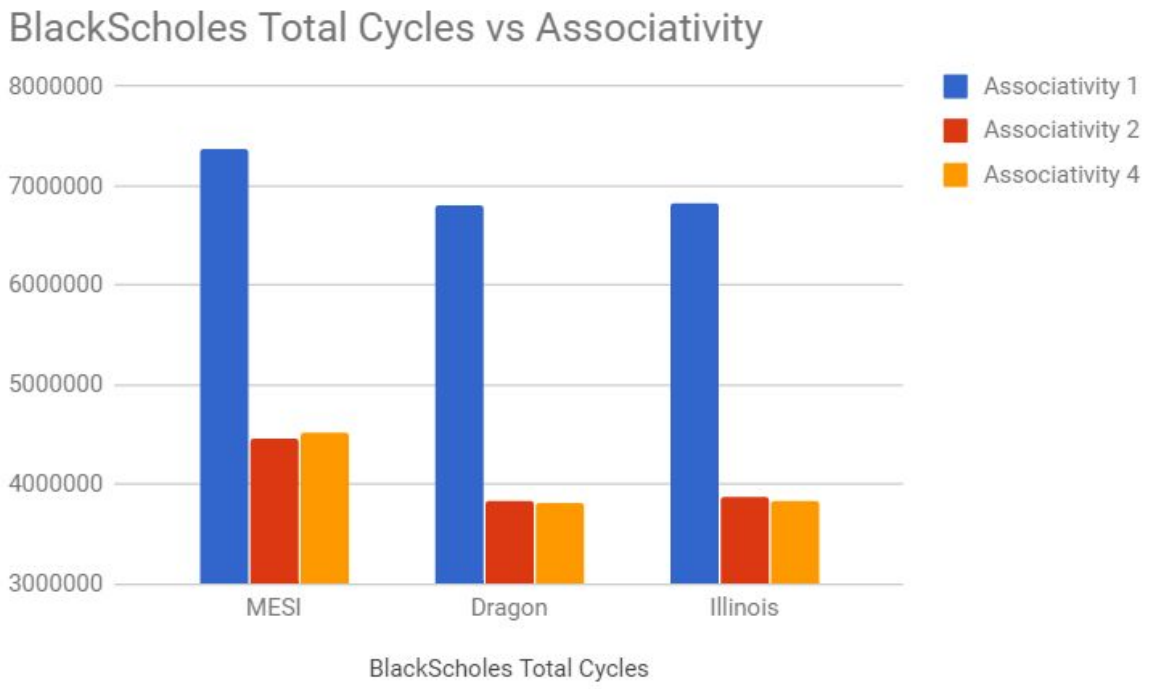
Quantitative Analysis

Pre-benchmark

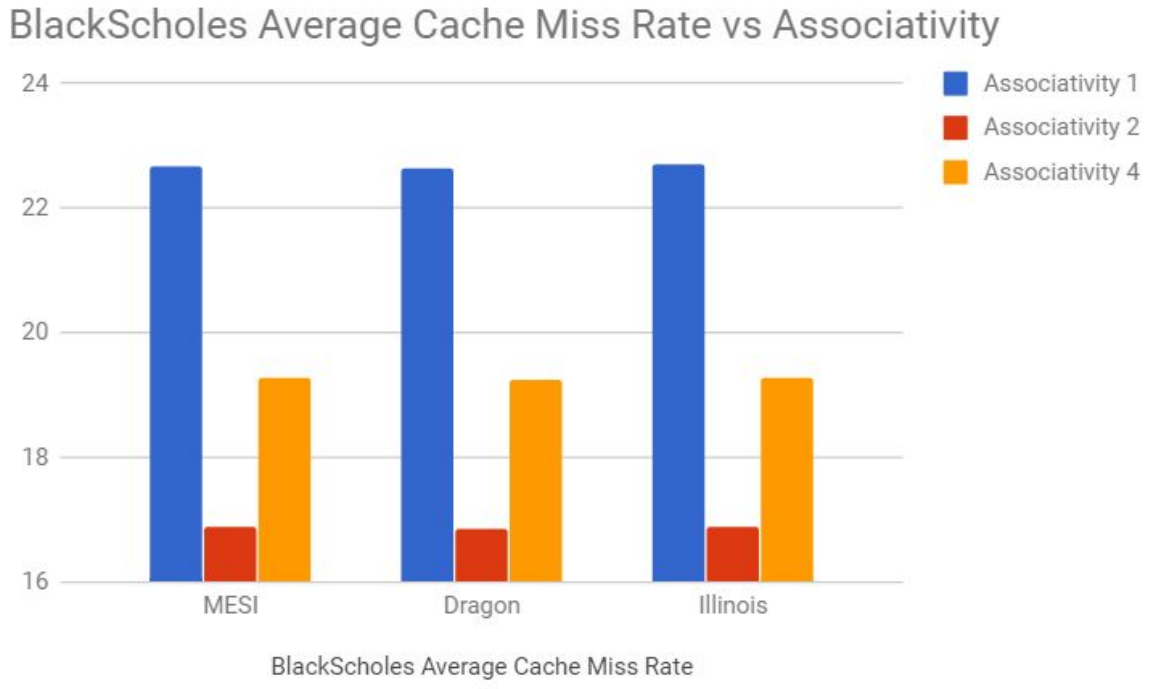
Due to the limitation of our Haskell implementation in terms of memory efficiency, running the entire trace file of more than 4 million traces caused the process to be terminated due to insufficient memory. Therefore, in order to still be able to benchmark the different protocols, we decided to cut down the trace files into 100,000 traces each. As mentioned above, this will still maintain a fair comparison between the different protocols since they're running on the same trace files, just that they may not be fully representative of each benchmark.

Benchmarks Results and Analysis

BlackScholes (Associativity = 1 or 2 or 4)

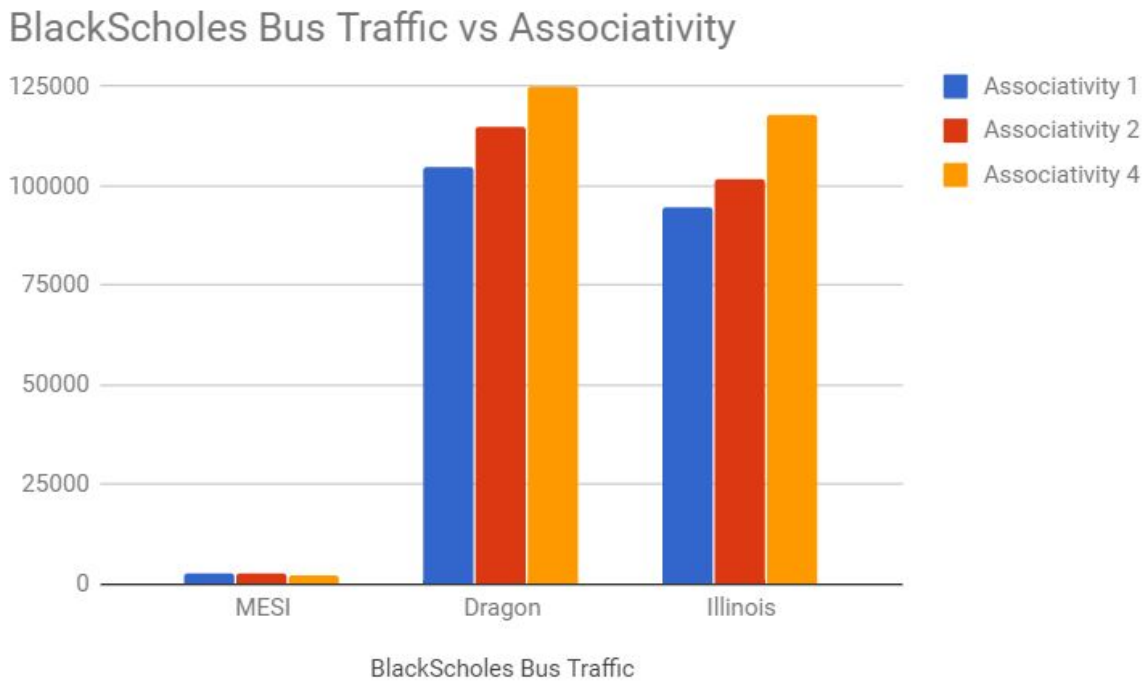


Associativity increases, total cycles drops (significantly from 1 → 2 but not 2 → 4). Dragon wins out Illinois barely in all cases.



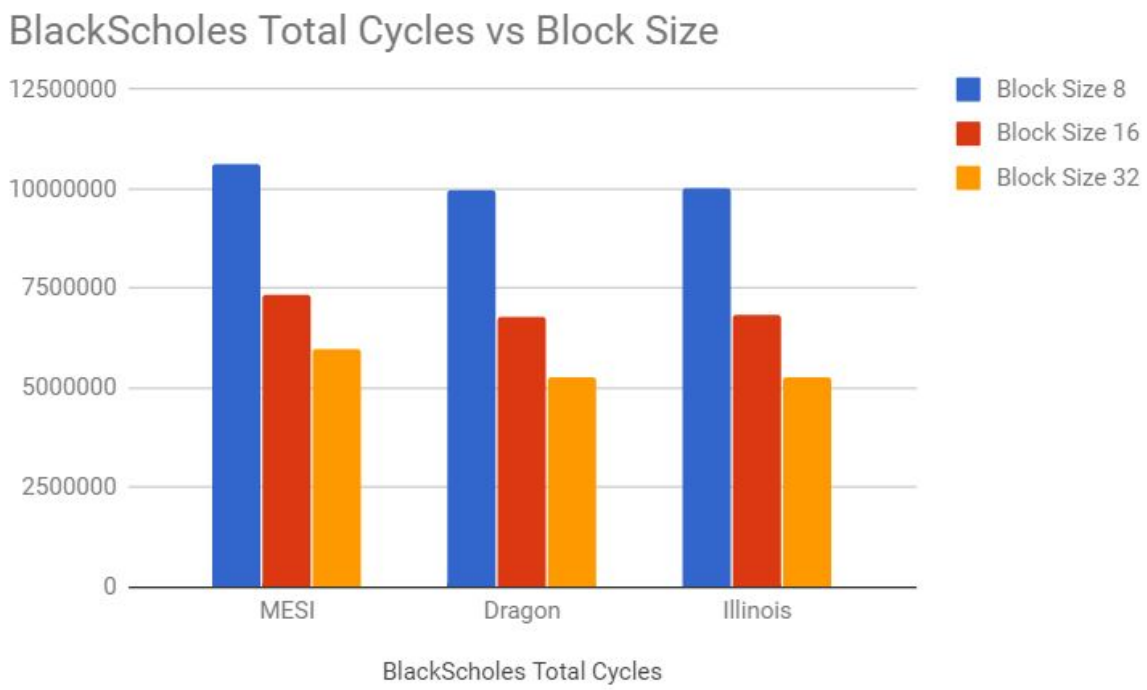
Average cache miss rate on all protocols seems to drop on associativity of 2, this is due to less conflict misses. Associativity of 4, however, seems to not reduce

the miss rate as much, most probably because associativity of 4 will cause the number of cache sets to drop significantly, mapping too many memory addresses to the same cache set which eventually causes more conflict misses. No clear difference between protocols here.

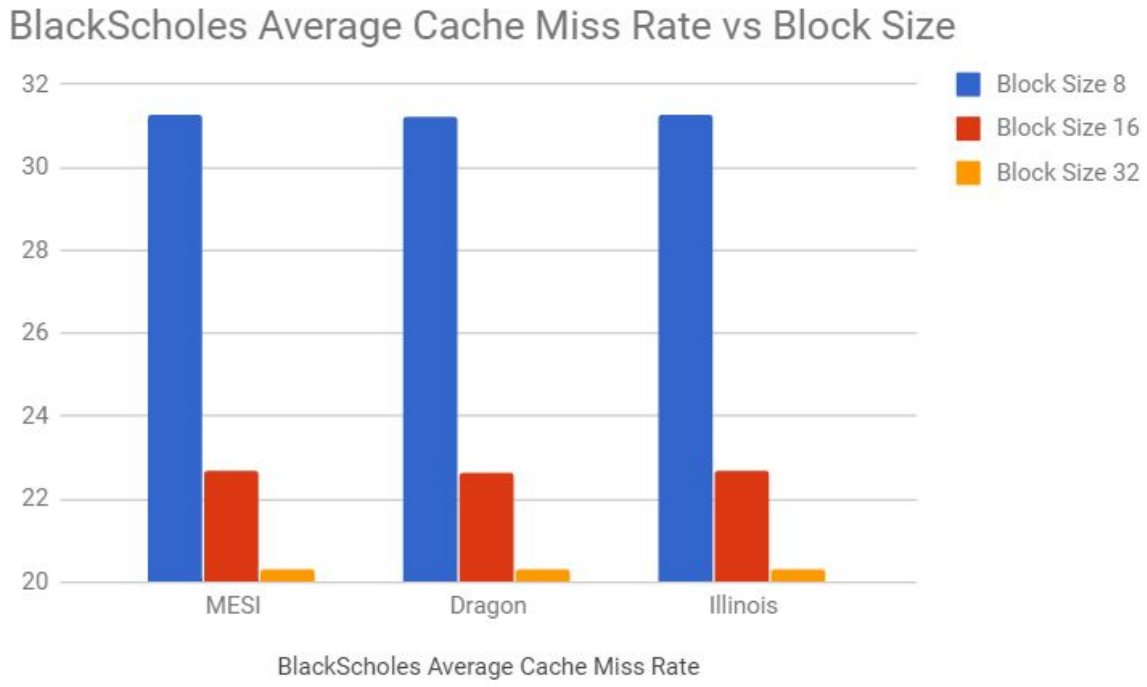


As expected MESI produces the minimum bus traffic since only flushes count. As associativity increases, for MESI, bus traffic *decreases* (because less evictions occur) but *increases* for Dragon and Illinois as there is a higher chance that another cache will not have evicted a block that you want, meaning more bus transfers occur to bring the block to you as opposed to memory reads.

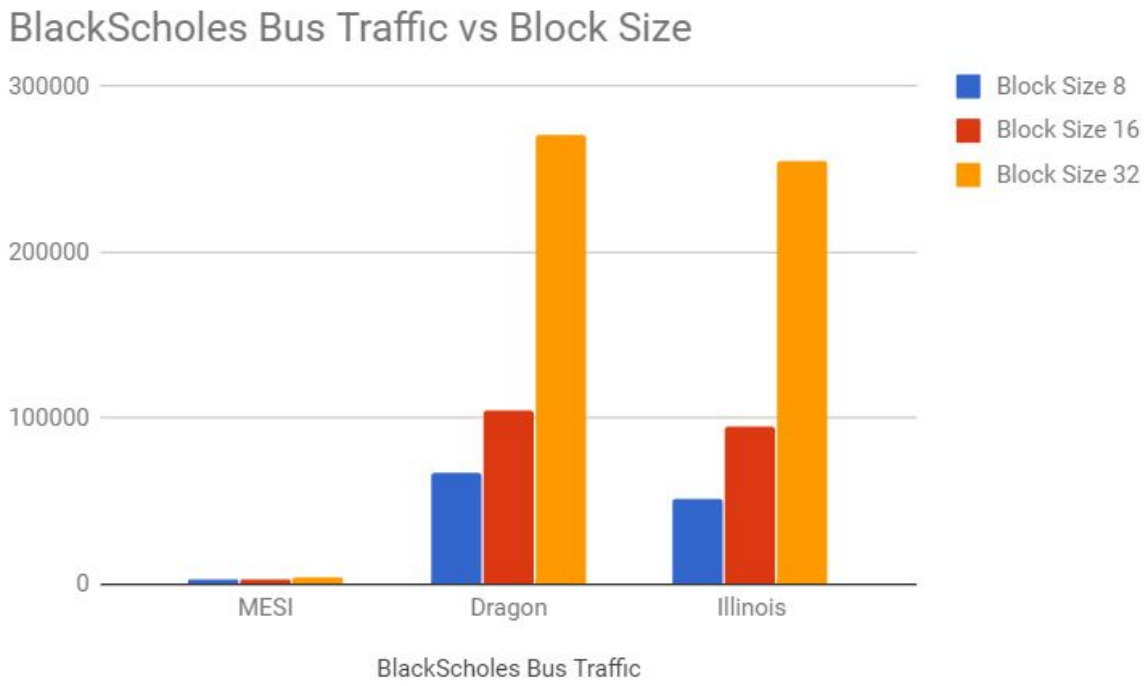
BlackScholes (Block Size = 8 or 16 or 32)



The total cycles drops across all protocols when block size is increased due to higher spatial locality (means that blackscholes has high spatial locality code/more array accesses). MESI tends to take more cycles than both Dragon and Illinois due to the fact that MESI needs to do more memory reads. Other than that, no clear distinction between the Dragon and Illinois protocol.

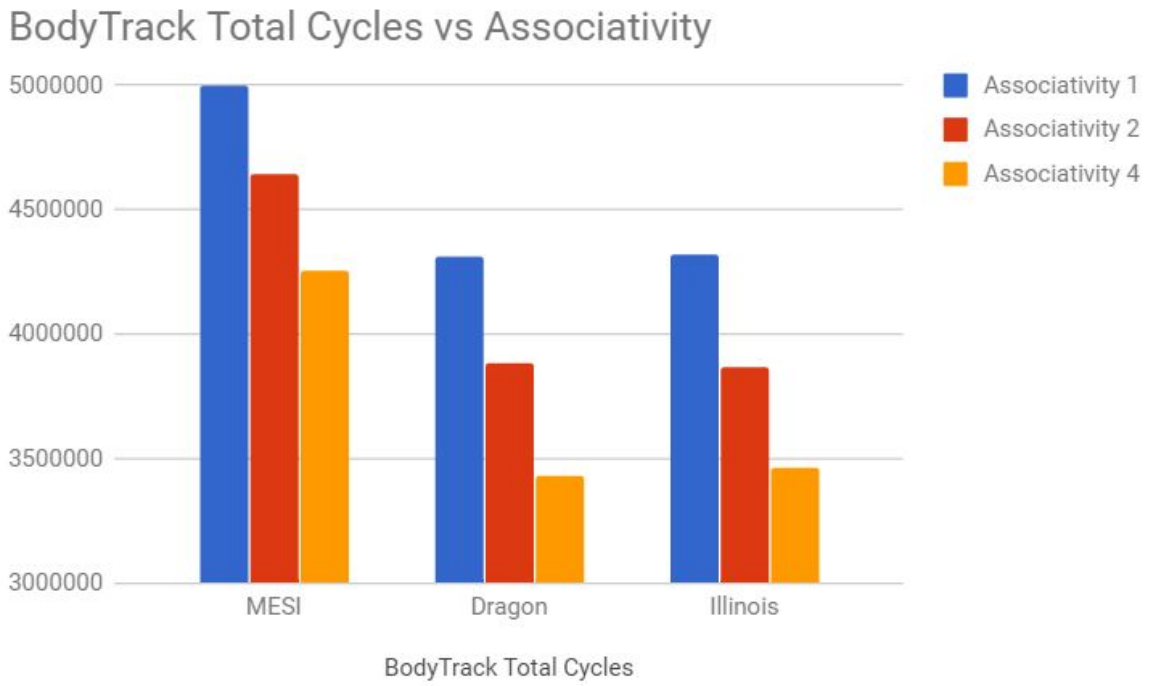


The cache miss rate drops across all protocols as block size increases since more words are brought into the cache per cache read - this increases the chance of a hit due to spatial locality which as stated above seems to be a feature of blackscholes.



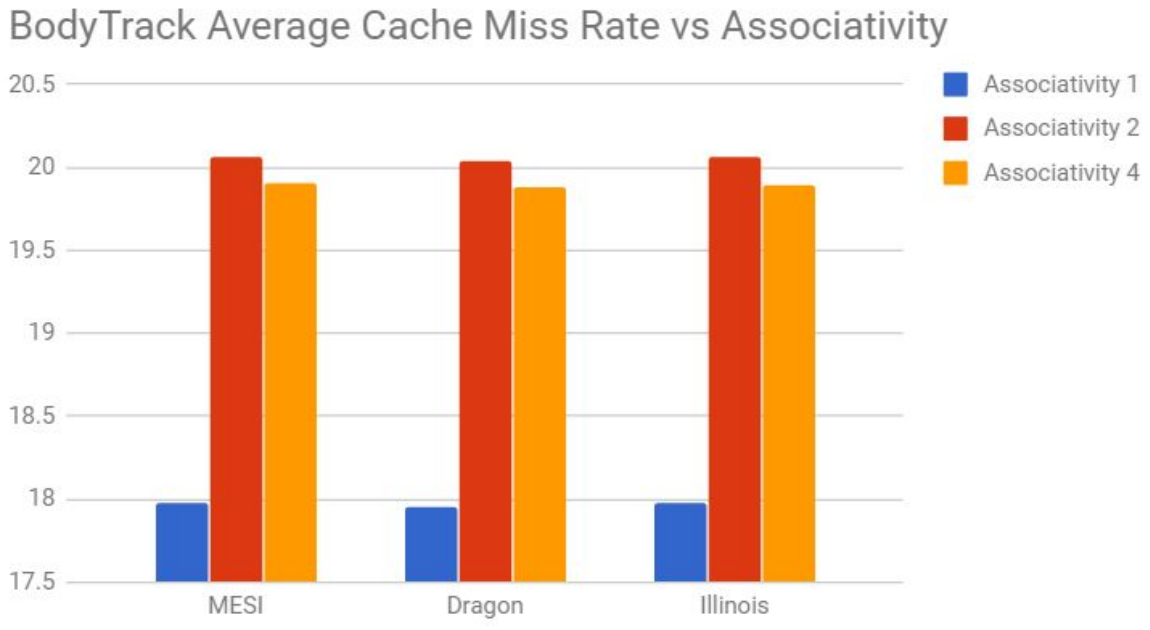
The bus traffic increases across all protocols as block size increases as more words need to be propagated through the bus whenever an update happens. MESI, however, is barely affected due to the fact that bus traffic in MESI only occurs whenever an M block gets invalidated and written back to memory. Dragon has slightly more bus traffic than Illinois here as Illinois would still invalidate other caches on write, giving less chances of other caches having a copy of the required data.

BodyTrack (Associativity = 1 or 2 or 4)



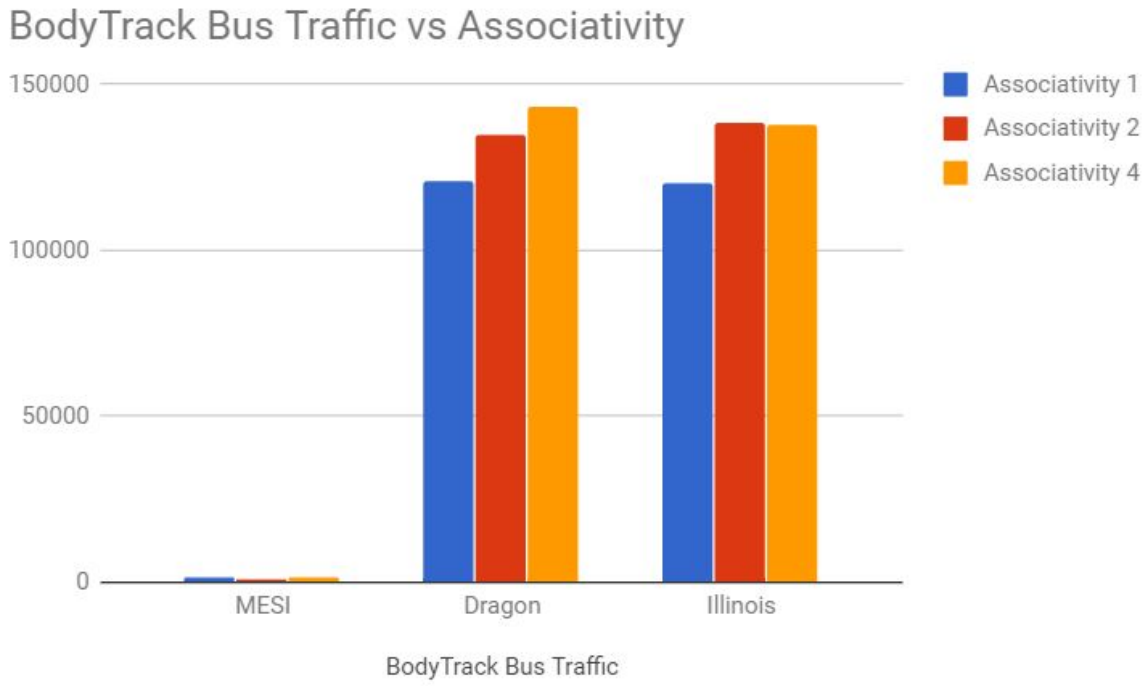
BodyTrack Total Cycles

Total cycles tend to decrease on all protocols when associativity is increased due to less conflict misses (more buffer to store data in a single block ID). Contrary to BlackScholes however, the total cycles keep decreasing sharply as associativity goes from 2 to 4. This indicates that the benchmark does not react to reduced number of cache sets negatively, indicating that more data have the same block ID than blackscholes.



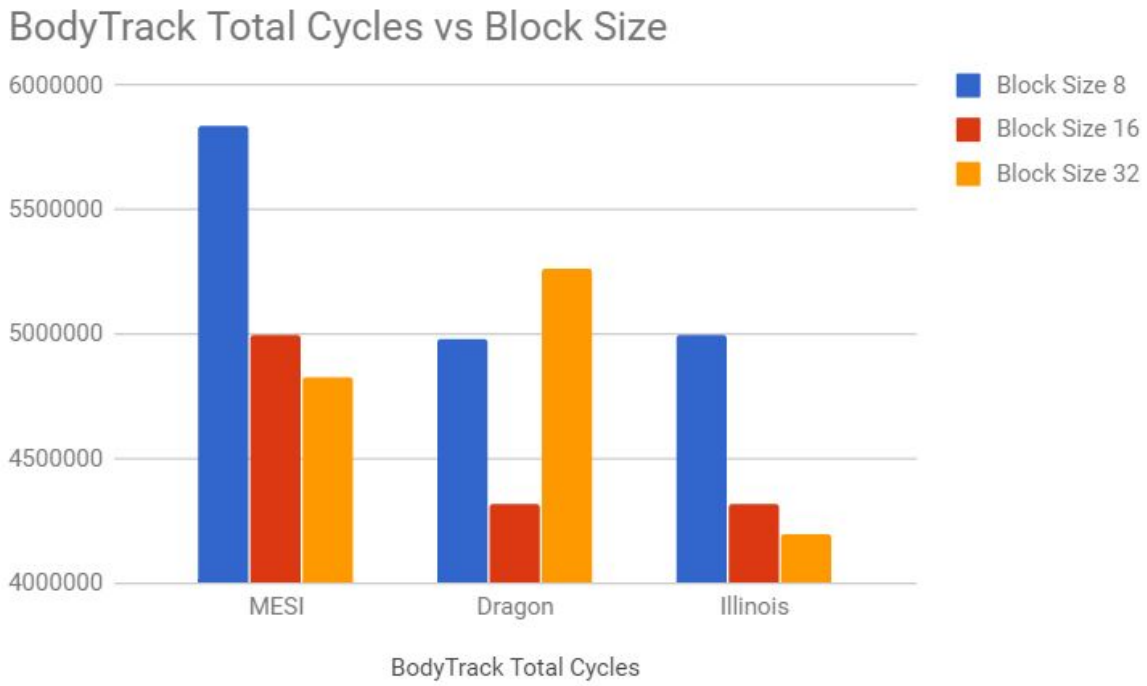
BodyTrack Average Cache Miss Rate

The average cache miss rate increases on non-direct mapped cache, indicating that the benchmark has higher tendency to cause conflict miss when associativity is high. No clear difference between the 3 protocols here.

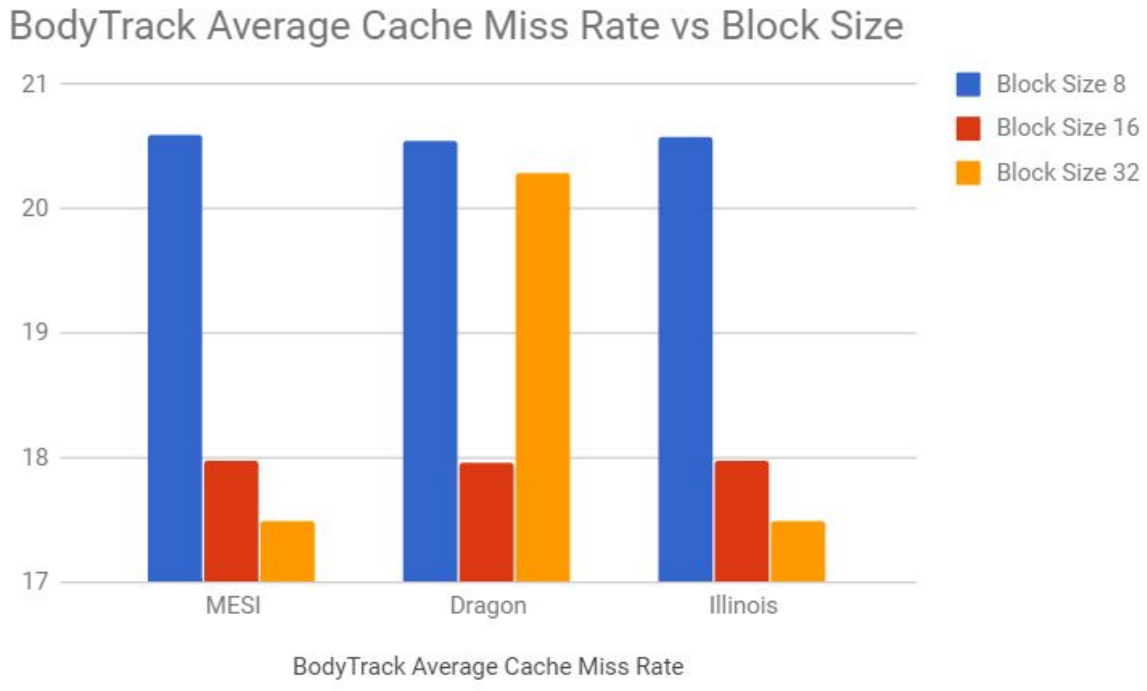


As expected, the bus traffic increases as associativity increases due to higher tendency for foreign caches to have the data that a cache currently needs. Again, this does not have a major effect on MESI as by nature MESI protocol does not have a lot of bus traffic (only due to write-back of M blocks on invalidation). Dragon and Illinois seem to have about the same amount of bus traffic on each associativity.

BodyTrack (Block Size = 8 or 16 or 32)

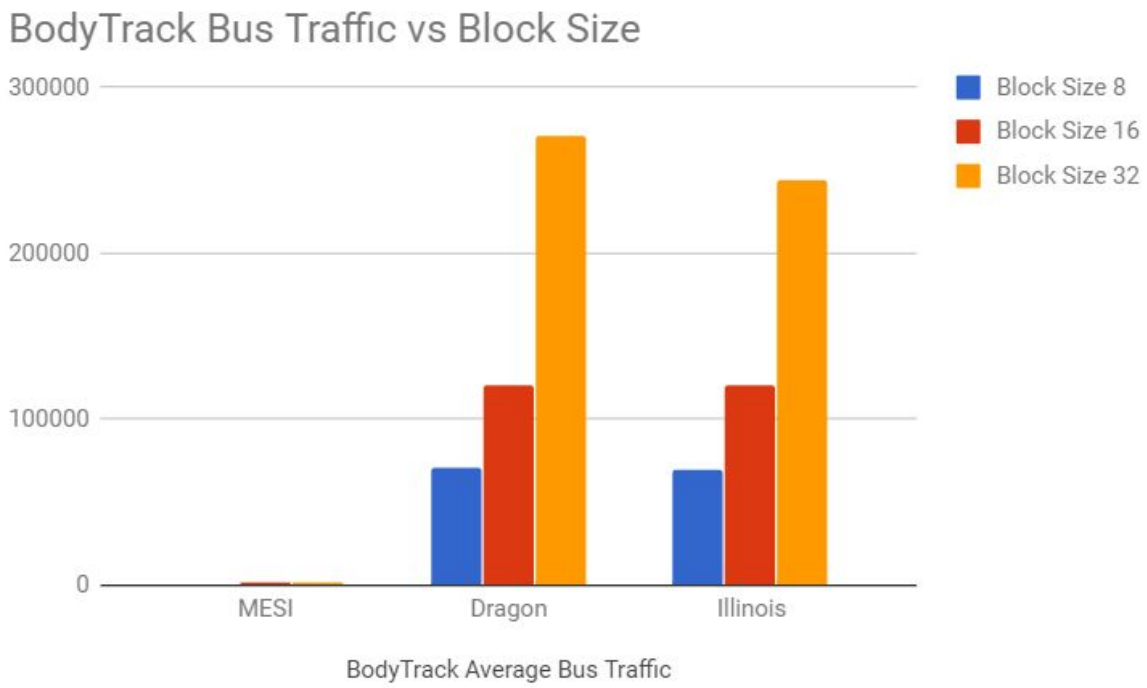


Increasing block size to 16 improves the total cycles of all protocols, but increasing again to 32 causes a downgrade of performance on Dragon. This might be because in Dragon the cost of updating other caches increases when block size is large, eventually causing a slow-down on the program. Illinois protocol seems to be a great candidate here.



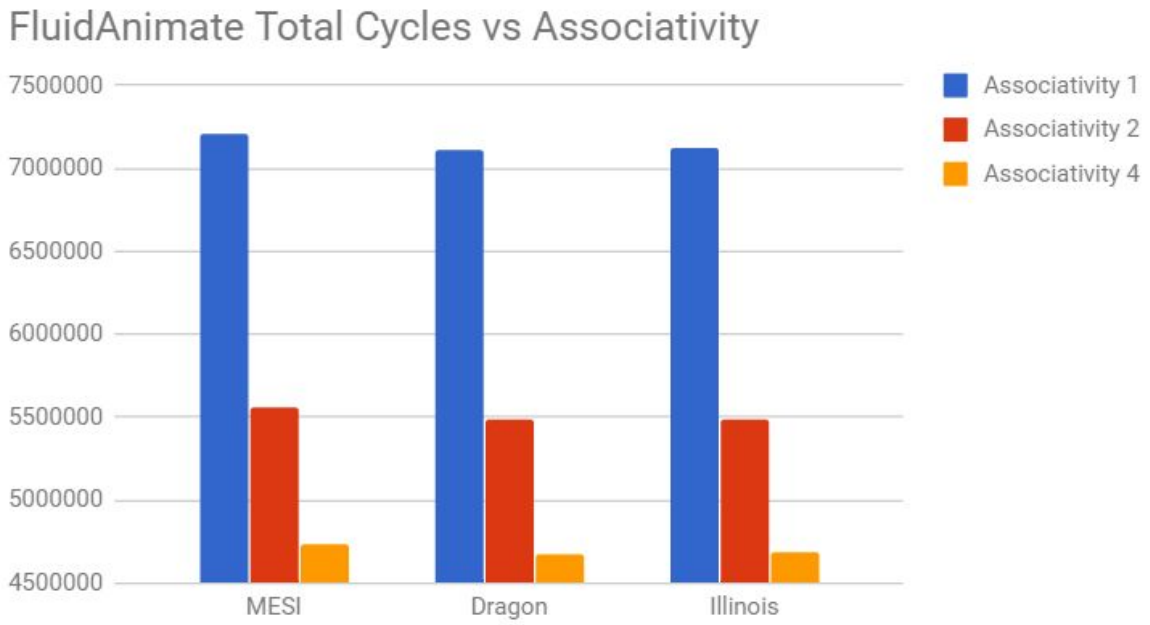
The average cache miss rate has similar trend with total cycles when block size is changed. No new observations here, aside from MESI being identical to Illinois

in terms of cache miss rate (which makes sense as Illinois is just extended MESI)



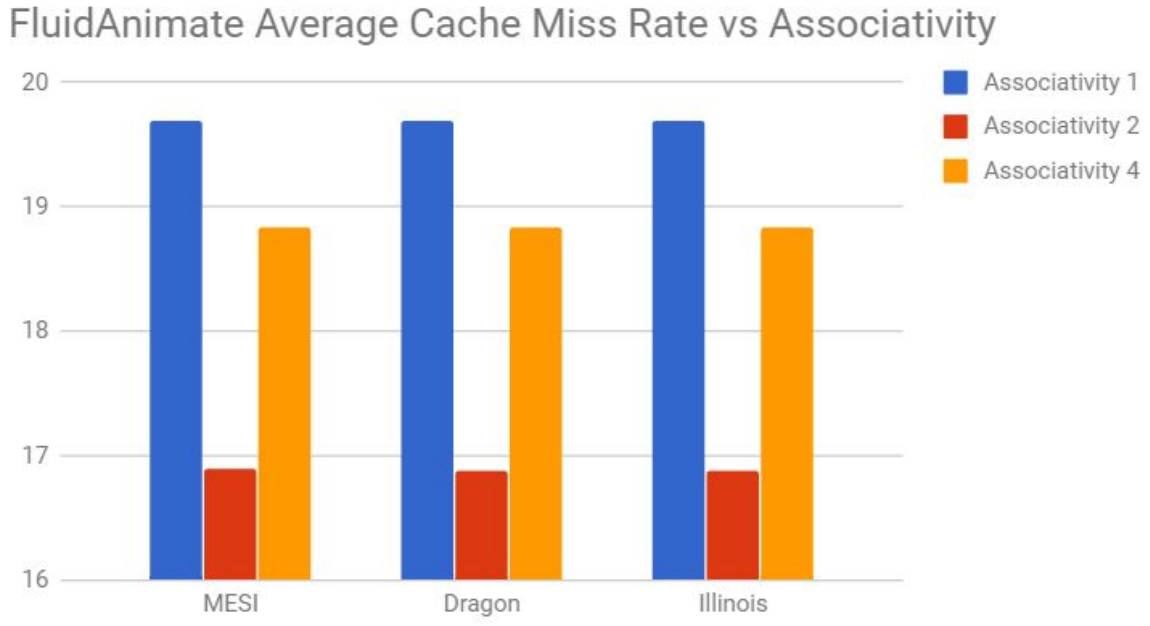
The bus traffic follows a similar trend as usual, with higher block size increasing bus traffic as more words need to be forwarded through the bus (MESI bus traffic is low due to bus traffic only occuring on write-back of M state during invalidation). Dragon and Illinois have similar amounts of bus traffic, but on block size 32, Dragon seems to have more traffic due to having more update mechanisms compared to Illinois.

FluidAnimate (Associativity = 1 or 2 or 4)



FluidAnimate Total Cycles

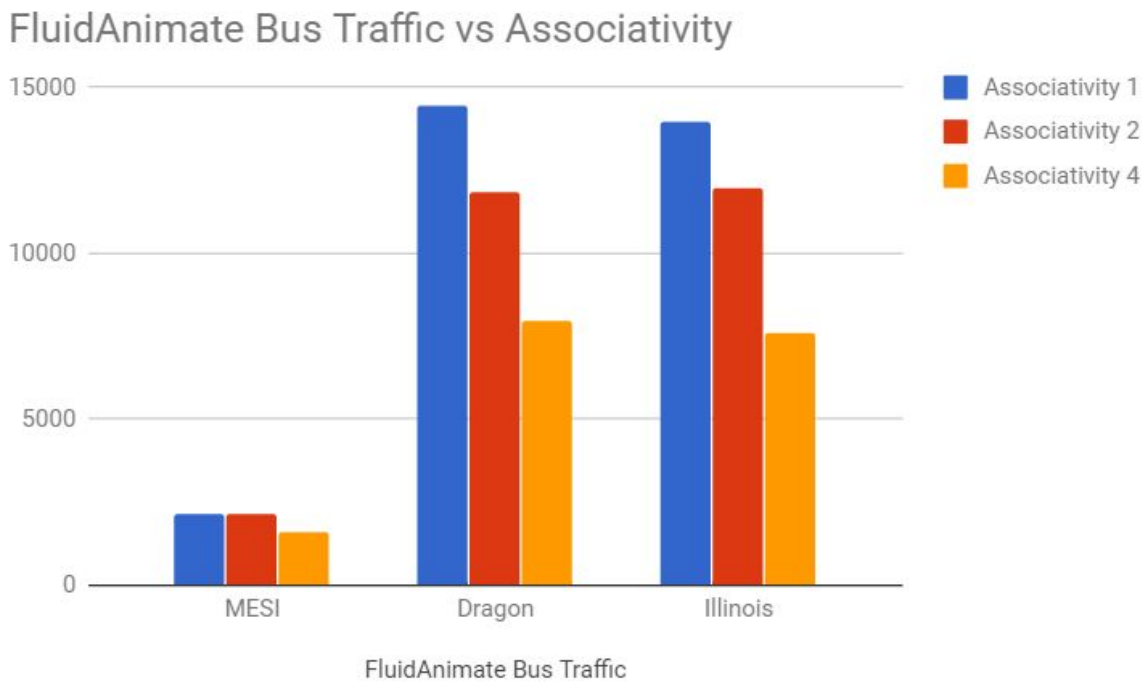
With decreasing associativity, FluidAnimate's total cycles decreases at roughly the same rate across all protocols. Dragon once again wins out by a very small margin of under 10,000 cycles. The drop in number of cycles is very sharp from associativity 1 → 2 vs. 2 → 4. This means that the reduction in number of cache sets affects FluidAnimate more than BodyTrack in this case.



FluidAnimate Average Cache Miss Rate

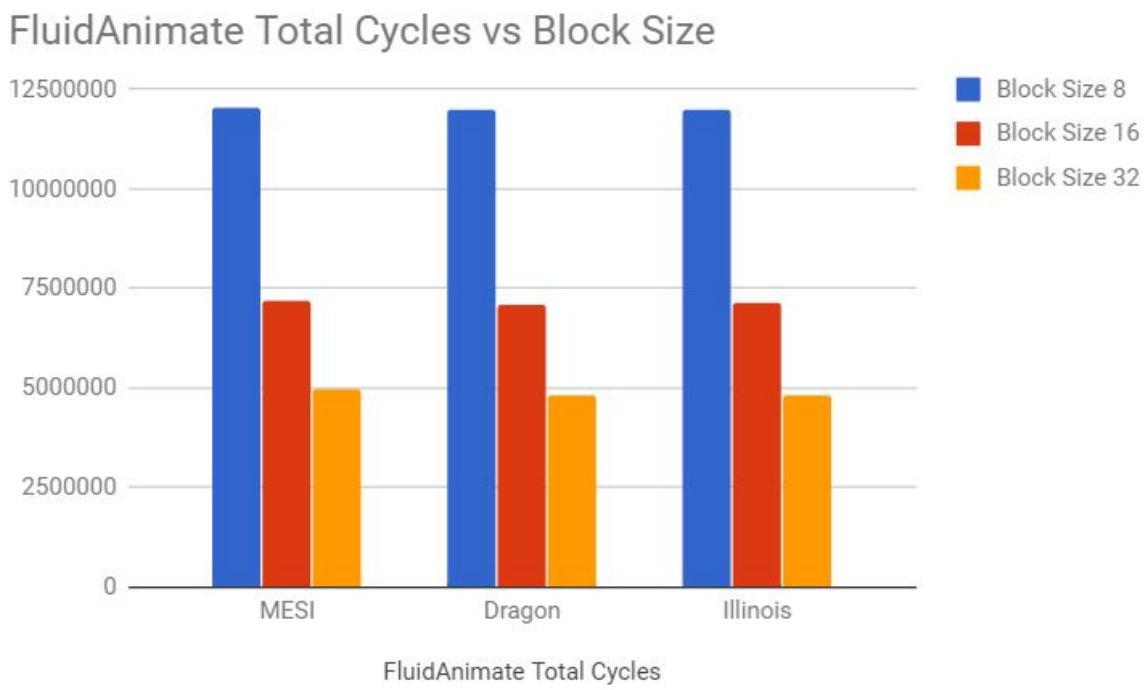
The average cache miss rate drops with an associativity of 2 but increases again with an associativity of 4. This indicates that the gains from decreasing conflict

misses with an associativity of 2 are outweighed by the effects of cache set reduction.

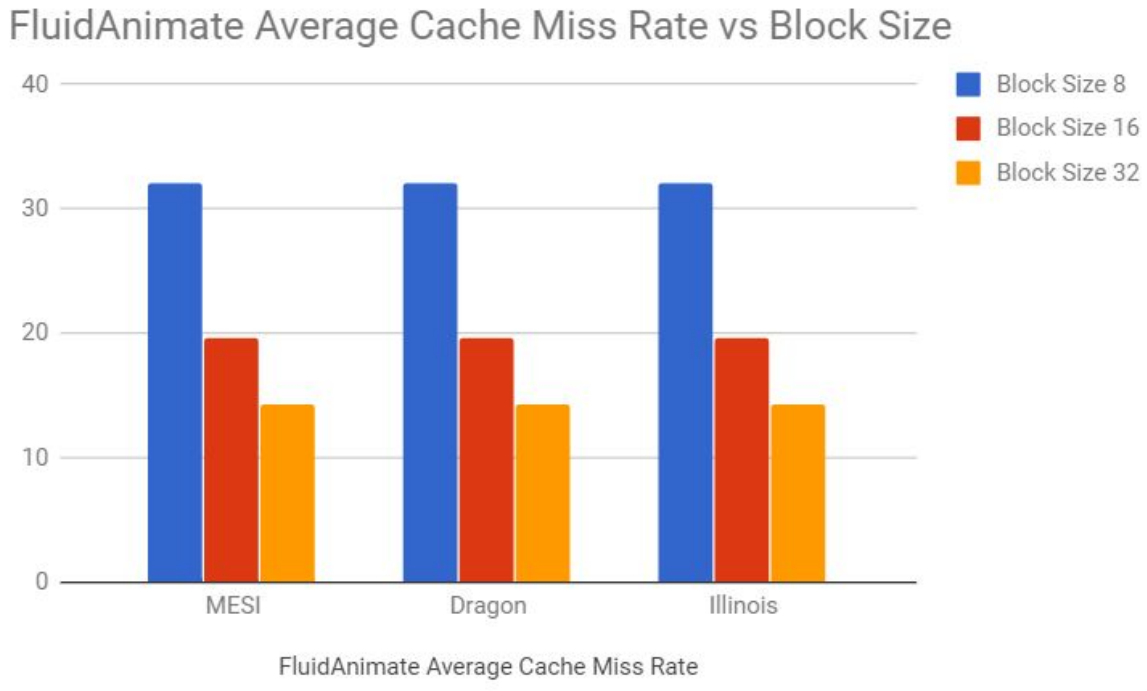


The bus traffic here seems to decrease with increasing associativity across all protocols. This likely means that given the cache miss rates above, with decreasing conflict misses., more data is being found in local caches while sharers are not modifying the data as opposed to having to receive many new blocks across the bus.

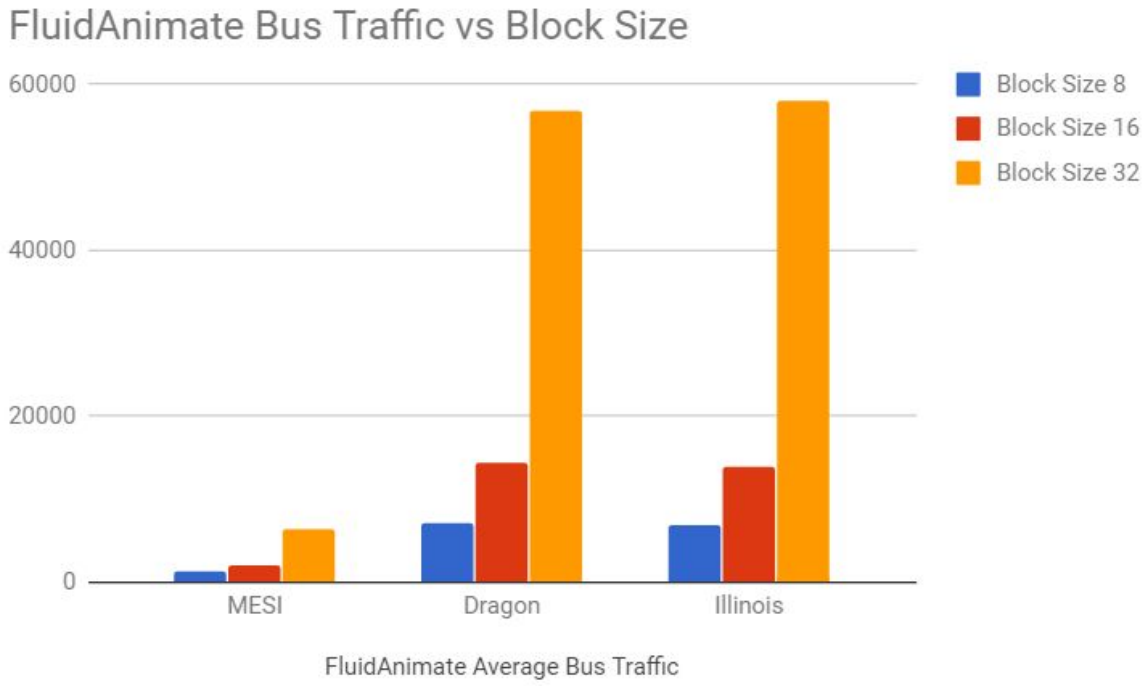
FluidAnimate (Block Size = 8 or 16 or 32)



All protocols behave similarly here; with increasing block size we get decreasing cycles. Once again this indicates that spatial locality is being taken advantage of well and that there is still more to be exploited.



Once again with increasing block size we get decreasing cache miss rates - probably the main causal factor of our decreased cycles as the memory needs to be hit less.



Finally we observe that bus traffic increases again with block size, as more data must be transmitted across the bus for each cache to cache transfer. What is interesting is that there is a very sharp increase in the bus traffic between 16 and 32 sized blocks, probably because the decrease in cache miss rate no longer counteracts the amount of data being pushed across the bus as well as it did for size 16 blocks.

Conclusion

In BlackScholes, Dragon/Illinois are leading in terms of total cycles by quite a margin, but MESI has a lot less bus traffic. The best candidate is arguable depending on whether total cycles is the critical measure or not. Limited bus traffic would mean less power required to operate the system, a much more important metric today in large supercomputers.

In BodyTrack, MESI requires a lot more cycles than Dragon or Illinois, to the point that it is no longer worth using even though the bus traffic is low. Between Dragon and Illinois, Illinois seems to be the better due to lower bus traffic, comparable total cycles, and higher potential of scaling on higher block sizes (>16 bytes).

In FluidAnimate, Dragon seems to be the best protocol to use in terms of total cycles, but MESI is not far behind in terms of total cycles with a lot less bus traffic. So MESI is the best candidate here due to less power consumption caused by low bus traffic.

Appendix I: Sample Raw Data Dump (Associativity = 1 across Benchmarks)

NB: All raw data is available on request.

BLACKSCHOLES_MESI

-----SIMULATION STATISTICS REPORT-----
Total Cycles: 7364734

PID: 0 | Compute Cycles: 290002 | Load Store Instructions: 50000 | Idle Cycles: 5888295 | Cache Miss Rate: 19.32%

PID: 1 | Compute Cycles: 270096 | Load Store Instructions: 50000 | Idle Cycles: 5813267 | Cache Miss Rate: 19.017999999999997%

PID: 2 | Compute Cycles: 237316 | Load Store Instructions: 50000 | Idle Cycles: 7127419 | Cache Miss Rate: 32.076%

PID: 3 | Compute Cycles: 229377 | Load Store Instructions: 50000 | Idle Cycles: 6065049 | Cache Miss Rate: 20.314%

Bus Traffic (Bytes): 2928
Bus invalidations/updates: 18970

Cache private accesses: 154670
Cache public accesses: 18889

BLACKSCHOLES_DRAGON

-----SIMULATION STATISTICS REPORT-----
Total Cycles: 6804000

PID: 0 | Compute Cycles: 290002 | Load Store Instructions: 50000 | Idle Cycles: 5611523 | Cache Miss Rate: 19.266%

PID: 1 | Compute Cycles: 270096 | Load Store Instructions: 50000 | Idle Cycles: 5461332 | Cache Miss Rate: 18.990000000000002%

PID: 2 | Compute Cycles: 237316 | Load Store Instructions: 50000 | Idle Cycles: 6566685 | Cache Miss Rate: 32.064%

PID: 3 | Compute Cycles: 229377 | Load Store Instructions: 50000 | Idle Cycles: 5769834 | Cache Miss Rate: 20.26%

Bus Traffic (Bytes): 104748
Bus invalidations/updates: 8514

Cache private accesses: 149128
Cache public accesses: 24455

BLACKSCHOLES_ILLINOIS

-----SIMULATION STATISTICS REPORT-----
Total Cycles: 6821515

PID: 0 | Compute Cycles: 290002 | Load Store Instructions: 50000 | Idle Cycles: 5438659 | Cache Miss Rate: 19.338%

PID: 1 | Compute Cycles: 270096 | Load Store Instructions: 50000 | Idle Cycles: 5385884 | Cache Miss Rate: 19.038%

PID: 2 | Compute Cycles: 237316 | Load Store Instructions: 50000 | Idle Cycles: 6584200 | Cache Miss Rate: 32.086%

PID: 3 | Compute Cycles: 229377 | Load Store Instructions: 50000 | Idle Cycles: 5602852 | Cache Miss Rate: 20.31%

Bus Traffic (Bytes): 94880
Bus invalidations/updates: 24731

Cache private accesses: 154276
Cache public accesses: 19282

BODYTRACK_MESI

-----SIMULATION STATISTICS REPORT-----
Total Cycles: 4997837

PID: 0 | Compute Cycles: 823255 | Load Store Instructions: 50000 | Idle Cycles: 4174583 | Cache Miss Rate: 17.572%

PID: 1 | Compute Cycles: 262064 | Load Store Instructions: 50000 | Idle Cycles: 4617534 | Cache Miss Rate: 18.945999999999998%

PID: 2 | Compute Cycles: 289934 | Load Store Instructions: 50000 | Idle Cycles: 4249850 | Cache Miss Rate: 17.291999999999998%

PID: 3 | Compute Cycles: 289386 | Load Store Instructions: 50000 | Idle Cycles: 4492769 | Cache Miss Rate: 18.11%

Bus Traffic (Bytes): 1472
Bus invalidations/updates: 6894

Cache private accesses: 133278
Cache public accesses: 37589

BODYTRACK_DRAGON

-----SIMULATION STATISTICS REPORT-----
Total Cycles: 4314846

PID: 0 | Compute Cycles: 823255 | Load Store Instructions: 50000 | Idle Cycles: 3491592 | Cache Miss Rate: 17.534%

PID: 1 | Compute Cycles: 262064 | Load Store Instructions: 50000 | Idle Cycles: 3907812 | Cache Miss Rate: 18.938%

PID: 2 | Compute Cycles: 289934 | Load Store Instructions: 50000 | Idle Cycles: 3588480 | Cache Miss Rate: 17.266000000000002%

PID: 3 | Compute Cycles: 289386 | Load Store Instructions: 50000 | Idle Cycles: 3803379 | Cache Miss Rate: 18.088%

Bus Traffic (Bytes): 120928
Bus invalidations/updates: 7801

Cache private accesses: 132019
Cache public accesses: 38886

BODYTRACK_ILLINOIS

-----SIMULATION STATISTICS REPORT-----
Total Cycles: 4316617

PID: 0 | Compute Cycles: 823255 | Load Store Instructions: 50000 | Idle Cycles: 3493363 | Cache Miss Rate: 17.566000000000003%

PID: 1 | Compute Cycles: 262064 | Load Store Instructions: 50000 | Idle Cycles: 3924940 | Cache Miss Rate: 18.944%

PID: 2 | Compute Cycles: 289934 | Load Store Instructions: 50000 | Idle Cycles: 3641674 | Cache Miss Rate: 17.291999999999998%

PID: 3 | Compute Cycles: 289386 | Load Store Instructions: 50000 | Idle Cycles: 3810389 | Cache Miss Rate: 18.102%

Bus Traffic (Bytes): 120272
Bus invalidations/updates: 14372

Cache private accesses: 133304
Cache public accesses: 37574

FLUIDANIMATE_MESI
-----SIMULATION STATISTICS REPORT-----
Total Cycles: 7202651

PID: 0 | Compute Cycles: 317851 | Load Store Instructions: 50000 | Idle Cycles: 5516771 | Cache Miss Rate: 19.07%

PID: 1 | Compute Cycles: 273866 | Load Store Instructions: 50000 | Idle Cycles: 2422507 | Cache Miss Rate: 7.448%

PID: 2 | Compute Cycles: 243401 | Load Store Instructions: 50000 | Idle Cycles: 6959251 | Cache Miss Rate: 32.306000000000004%

PID: 3 | Compute Cycles: 226651 | Load Store Instructions: 50000 | Idle Cycles: 5704923 | Cache Miss Rate: 19.968%

Bus Traffic (Bytes): 2160
Bus invalidations/updates: 28477

Cache private accesses: 134920
Cache public accesses: 54137

FLUIDANIMATE_DRAGON

-----SIMULATION STATISTICS REPORT-----
Total Cycles: 7112161

PID: 0 | Compute Cycles: 317851 | Load Store Instructions: 50000 | Idle Cycles: 5444048 | Cache Miss Rate: 19.07%

PID: 1 | Compute Cycles: 273866 | Load Store Instructions: 50000 | Idle Cycles: 2386659 | Cache Miss Rate: 7.448%

PID: 2 | Compute Cycles: 243401 | Load Store Instructions: 50000 | Idle Cycles: 6868761 | Cache Miss Rate: 32.302%

PID: 3 | Compute Cycles: 226651 | Load Store Instructions: 50000 | Idle Cycles: 5634352 | Cache Miss Rate: 19.964000000000002%

Bus Traffic (Bytes): 14472
Bus invalidations/updates: 936

Cache private accesses: 135078
Cache public accesses: 53983

FLUIDANIMATE_ILLINOIS

-----SIMULATION STATISTICS REPORT-----
Total Cycles: 7121933

PID: 0 | Compute Cycles: 317851 | Load Store Instructions: 50000 | Idle Cycles: 5457440 | Cache Miss Rate: 19.07%

PID: 1 | Compute Cycles: 273866 | Load Store Instructions: 50000 | Idle Cycles: 2389518 | Cache Miss Rate: 7.448%

PID: 2 | Compute Cycles: 243401 | Load Store Instructions: 50000 | Idle Cycles: 6878533 | Cache Miss Rate: 32.306000000000004%

PID: 3 | Compute Cycles: 226651 | Load Store Instructions: 50000 | Idle Cycles: 5637658 | Cache Miss Rate: 19.966%

Bus Traffic (Bytes): 13936
Bus invalidations/updates: 29240

Cache private accesses: 134841
Cache public accesses: 54217