CS4223 Multi-Core Architecture

Assignment 2 Report

1 Introduction

This report documents our work on the cache coherence simulator for Assignment 2. We detail our assumptions and implementation in the first section, followed by a short literature review on enhancements to the MESI protocol. We then analyse the results of the experiments conducted on the simulator in the quantitative analysis section.

2 Implementation

2.1 Assumptions

In addition to the assumptions stated in the assignment description, we have made the following assumptions in our implementation and measurements.

Implementation

- For MESI, data on miss is supplied by other caches (cache-to-cache sharing) if available, even if the time incurred from cache-to-cache transfer is larger than accessing memory. Furthermore, only one cache will be chosen by the bus to respond.
- For cache-to-cache sharing in all protocols, all caches involved in the sharing are hogged. The hogged caches are unable to service cache requests from the processor.
    - MESI: The cache supplying data on another cache's miss.
    - Dragon. All caches receiving a busUpd.

Metrics

- Cache hits are defined as reading/writing from a cache block in the following states:
    - MESI: M, E, S
    - Dragon: All states
- Private data is defined as cache blocks in the following states:
    - MESI: M, E
    - Dragon: M, E
- Shared data is defined as cache blocks in the following states:
    - MESI: S
    - Dragon: Sm, Sc

2.2 Program Overview

The specific requirements and instructions to run the simulator can be found in the repository README. Our simulator is implemented in Java, which was chosen for three key reasons:

- Object-oriented paradigm: Suitable for modeling the various hardware components of a multi-core system.
- Statically-typed: Greater ease of development with type-checking to catch errors early.
- Portable: Java is pre-installed on many platforms and the bytecode can be run without compilation on the platform.

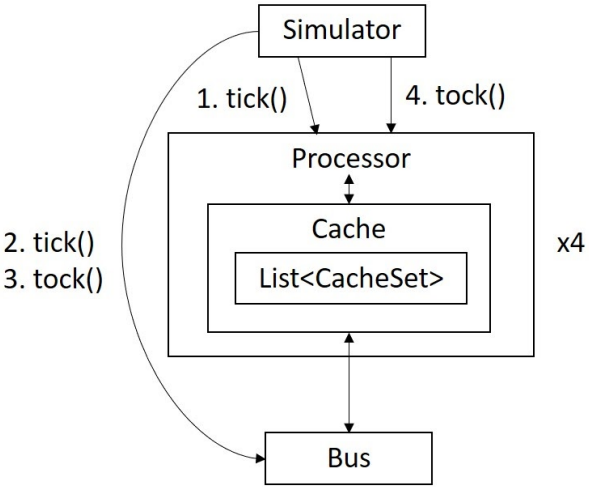

Figure 1. Simulator Architecture Diagram

The overall architecture of the program is depicted in the figure above. On invocation, the simulator's main function performs the following loop until all processors report that they have completed trace simulation:

1. processors.forEach(processor.tick());
2. bus.tick();
3. bus.tock();
4. processors.forEach(processor.tock());

2.3 Processor

Processor States

- READY: Will onboard the next instruction (or go to DONE) on the next tick()
- COMPUTE: Doing computation (instruction type 2)
- WAITCACHE: Waiting for cache
- DONE: Completed all instructions.

Processor Methods

- tick(): If state is READY, loads up the next instruction. Always calls cache.tick() at the end regardless of whether an instruction was loaded.
    - If the instruction is compute, set computeRemaining to the number of compute cycles, change state to COMPUTE.
    - If the instruction is read/write, call cache.read/write(address) respectively, change state to WAITCACHE.
- tock(): If state is COMPUTE, decrement computeRemaining and increment the processorComputeCycles stat. If state is WAITCACHE, increment the processorIdleCycles stat.
- unstall(): Called by the cache once the read/write operation is complete. Changes the state from WAITCACHE to READY.

2.4 BusTransaction

Represents a transaction to be put on the bus. Holds the following:

- transition: Transaction type under the enum Transition.
- address
- size
- shared: Whether the cache block is shared. Only applies to responses.

2.5 Transition

Bus transaction types are as in the protocols, with the following variations from the types covered in class:

- BusRdX: In our implementation, this is exclusively for I -> M transition (MESI) and indicates the need to generate bus traffic for fetching the data from memory/another cache.
- BusUpgr: Invalidates the block on other caches so that the cache requesting this transaction can go from S to M state (MESI) without generating bus data traffic.
- FlushOpt: A cache holding a block in S state (MESI) will respond to a snooped BusRd/X with a FlushOpt to indicate its willingness to offer the block for cache-to-cache transfer.

2.6 Cache

Cache is an abstract class modelling an L1 cache. It is inherited by concrete subclasses, each implementing a specific protocol by defining the various transitions between states. For example, MesiCache implements MESI.

Cache States

- READY: Available to accept a read() or write() from the processor
- PENDING_WRITE: Will do prWr() on the next tick() (unless hogged)

- WRITING_WAITBUS: The operation in progress is a write, and waiting to put a transaction on the bus.
- WRITING_PENDING_FLUSH: The operation in progress is a write, and an eviction Flush has been put on the bus and is being waited on to complete.
- WRITING: The operation in progress is a write, and some non-Flush transaction has been put on the bus and is being waited on to complete.

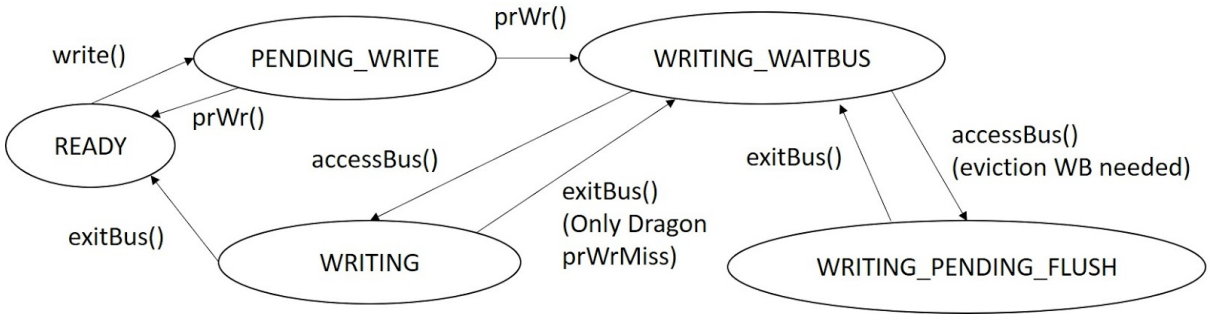

Figure 2. State Transition Diagram for Cache

The states and transitions for reading are the same except with the appropriate name changes e.g. PENDING_READ.

The exceptional transition from WRITING to WRITING_WAITBUS is for Dragon prWrMiss, where the cache does a BusRd, finds that the block is shared and subsequently has to perform a BusUpd. This does not apply to other protocols, and this transition does not exist in the state transition model for cache reads.

Cache Methods

Concrete Methods

- write(address): Change state from READY to PENDING_WRITE, and puts the address in pendingAddress.
- read(address): Similar, for reading.
- tick(): If cache is hogged, does nothing. If cache is in PENDING_WRITE, calls the protocol specific processor write function, similar for PENDING_READ. Otherwise does nothing.

Abstract Methods

- prRd/Wr(): Processor read/write function called from tick(). Transitions cache state to READY if operation can be done without the bus. Otherwise calls bus.reserve().
- accessBus(): Called by the bus. Dynamically generates the appropriate bus transaction to be put on the bus.
    - Such dynamic generation circumvents certain problematic scenarios. For example, while a cache is waiting in the bus's queue to perform a conflict/capacity miss eviction, it may snoop an invalidation from another cache that was closer to the front in the queue, which eliminates the need to evict the block. When accessBus() is finally called, the method will detect that

there is indeed space for the new block being brought in, and respond with a BusRd instead of a Flush.

- exitBus(transaction): Called by the bus. The transaction is completed. The cache makes the appropriate block state changes as specified in the protocols, processor.unstall() is called and the cache returns to READY state.
    - There are certain exceptions, namely FLUSH, and Dragon write miss, as specified previously.
- snoop(transaction): Performs the necessary downgrading of the cache block if the cache does indeed contain it. If protocol is MESI, may return with FlushOpt or Flush.

2.7 Bus

There is a singular Bus class as each bus transaction works the same way across all protocols.

The bus has two states, READY and BUSY.

Object Fields

- requesterQueue: Contains, in FIFO order, the caches that want to use the bus.
- primaryCycles: Countdown until the requesting cache should be replied to with the result of its transaction.
- secondaryCycles: Countdown until hogged caches can be unhogged.
- hogged: A set of caches which are being hogged for cache-to-cache data transfer or M->S writeback to memory.

Bus Methods

When bus.tick() is called, if the bus is BUSY, nothing happens. If the bus is READY, and there is a requester in requesterQueue, the bus calls requester.accessBus() to get the transaction and passes it to the appropriate handler function.

The handler functions generally do the following:

- Scan all caches other than the requester for the presence of the block. If any such cache holds the block then the shared flag in the reply will be set to true.
- Calls cache.snoop(txn) on each non-requester cache and collects the response transactions.
    - If no responses are received, primaryCycles = 100 representing memory fetch penalty (unless BusUpgr in which case set to 1 to resolve the same cycle).
    - If the responses consist of one or more FlushOpts, the cache with the smallest id is chosen to be hogged and both primaryCycles and secondaryCycles are set to the block size-dependent penalty as given in the notes.
    - If the response is a Flush (must be M state cache doing simultaneous flush to memory and cache-to-cache transfer) then the responder is chosen to be hogged and primaryCycles is set to the block size-dependent penalty, and secondaryCycles is set to max(100, primaryCycles).

- If BusUpd, and the block is indeed shared, choose all non-requester caches that hold the block to be hogged and set primaryCycles and secondaryCycles to the size-dependent penalty.
- Call cache.hog() on all caches previously chosen to be hogged.

bus.tick() then decrements primaryCycles and secondaryCycles. requester.exitBus(txn) is called once primaryCycles hits 0, and hogged.forEach(cache.unhog()) is called once secondaryCycles hits 0. Once primaryCycles and secondaryCycles are both non-positive, the bus transitions back to READY state.

2.8 Overall Flow

Putting all the pieces together, the overall flow of execution during a clock cycle is as such:

1. main calls processor.tick()
2. processor loads the next instruction if ready, accesses cache if needed
3. processor calls cache.tick()
4. cache does nothing if hogged, else calls prRd/Wr if pending.
    a. cache may request bus in prRd/Wr
5. main calls bus.tick()
6. bus, if ready, loads a requester if there is a requester in the queue.
    a. bus calls requester.accessBus() to get the transaction
    b. bus sets primary/secondaryCycles as appropriate
    c. bus calls hog() on processors involved in cache-to-cache transfer/memory writeback.
7. main calls bus.tock()
8. bus ticks down primary/secondaryCycle. Calls unhog() on hogged caches when secondaryCycle hits 0. Calls requester.exitBus(transaction) when primaryCycle hits 0.
    a. requester makes the appropriate state transitions for both the cache block and the cache as a whole based on the response received.
9. main calls processor.tock()
10. processor increments compute/idle cycles stats as appropriate, decrements computeCycles if doing computation.

2.9 BlockState

The BlockState enum contains the states for all the protocols. Each state is prefixed with the protocol it belongs to, for example MESI_MODIFIED. This was done for implementation expediency.

2.10 CacheSet

The CacheSet class represents one set-associative cache set. It is implemented with LinkedHashMap<Tag, BlockState> as it has FIFO iteration order. The following are some CacheSet methods which demonstrate how the LinkedHashMap is used:

- use(tag): Removes then puts back the tag so that it comes last in the iteration order.

- update(tag): Sets the state of the block at the given tag without first removing the tag. This maintains the tag's place in the iteration order as tag states could be updated due to a snooped transaction, which does not constitute a usage in the LRU replacement policy.
- getEvictionTargetTag(): Returns blocks.keySet().iterator().next(), which is the first tag in the FIFO iteration order, meaning the tag is the least recently used.

3 Advanced Task: MESI Enhancements

For the advanced task, we chose to explore enhancements to the MESI protocol.

MESIF

A possible enhancement of the MESI protocol is the addition of a Forward (F) state. In MESI, when a bus read is performed on a cache block that exists in S state on multiple caches, there are two possible ways the request is served, depending on the MESI implementation:

1. The block is fetched from main memory
2. Multiple sharing caches respond with the requested block (bombarding the requestor and bus with redundant responses)

Both these outcomes are undesirable, as it leads to unnecessary overhead in servicing the request. The F state acts as a special form of the S state; only one cache among sharing caches can hold a block in F state. A bus read on the cache block is responded to only by the cache in F state (a "designated responder"), preventing redundant responses from the other S state caches from flooding the bus.

However, as a cache may unilaterally decide to evict a block in F state, leaving no designated responders among S state caches, cache blocks might occasionally have to be fetched from memory even when the requested block exists on another cache. This is a downside of the MESIF protocol, which is partially mitigated by transferring the F state ownership to the most recent requestor of the cache block.

In our implementation of MESI, when a cache block is requested, only one cache would respond with the block, even if multiple caches share it. Thus, we do not expect the MESIF protocol to bring any performance benefits and did not choose to implement it.

MOESI (Chosen)

Another enhancement of the protocol is the introduction of an Owned (O) state. In environments where read and writes are frequently performed on a single cache block, a significant amount of time can potentially be spent on accessing memory. MOESI seeks to address this by removing the need to flush a cache block to memory whenever an M state transits to S state. The new O state is a distinguished S state - it indicates that a cache shares a valid copy of the block with other caches, but this copy moving around the caches is dirty and does not exist in memory. Only one cache among sharing caches can hold a cache block in O state. When the cache block in the O state is evicted or invalidated, it should be flushed to memory.

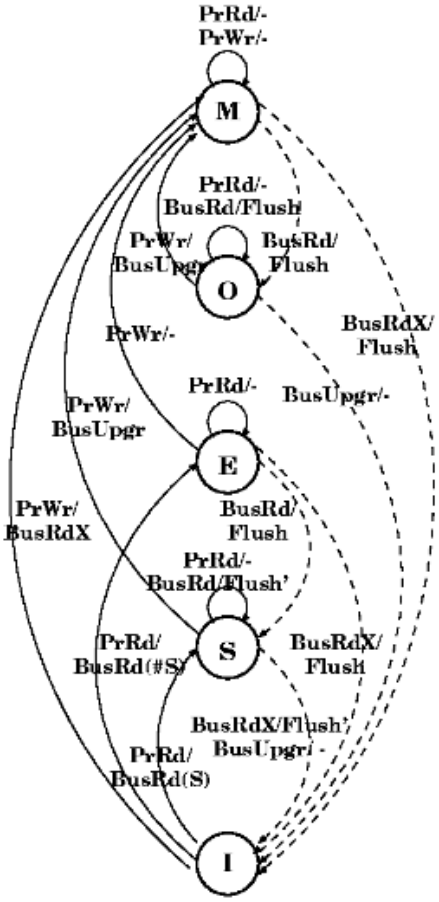

Figure 3. MOESI State Diagram[1]

We have chosen to implement this enhancement in our simulator program. Our implementation of the O state and the transitions match the state diagram in the figure above.

Read Broadcasting

Apart from adding new states, there exists other optimisations that have been proposed. One example is read broadcasting[2]. When a FlushOpt is performed, instead of just having the requesting cache receive the block, all processors will snoop the block. If the block exists in invalid state in the cache, it will be updated to S state with the new data. This is expected to happen when the block previously existed in the cache, but was invalidated by a write to the block in another cache. This helps improve temporal locality, because a later request to the block will be met with a hit instead of a miss. However, it has also been found that this approach could lead to contention between the bus and processor when accessing the cache's tag store. As the tag store and its ports are not modelled in our simulator, we have chosen not to implement this optimisation as our simulator may not accurately capture the tradeoff's impact on performance.

---

[1] "INTEGRATION AND EVALUATION OF CACHE COHERENCE ...." https://smartech.gatech.edu/bitstream/handle/1853/14065/suh_taeweon_200612_phd.pdf. Accessed 12 Nov. 2020.

[2] "An Evaluation of Snoop-Based Cache Coherence Protocols." https://hps.ece.utexas.edu/people/suleman/class_projects/pca_report.pdf. Accessed 13 Nov. 2020.

4 Quantitative Analysis

4.1 Workloads

	Instructions (Billions)			Parallelization Granularity	Working Set	Data Usage	
Program	**Total**	**Reads**	**Writes**			**Sharing**	**Exchange**
bodytrack	14.03	3.63	0.95	medium	medium	high	medium
blackscholes	2.67	0.68	0.19	coarse	small	low	low
fluidanimate	14.06	4.80	1.15	fine	large	low	medium

Figure 4. PARSEC Workload Traits

The PARSEC benchmarks and their traits are detailed in a technical report[3]. We reproduce the relevant traits in the figure above.

- Instructions: The instruction counts indicates how large the programs are and how many reads/writes are performed in total.
- Parallelization Granularity: While not a direct metric for data usage, coarse grained parallel programs are likely to share less data as individual threads execute larger tasks.
- Working Set: Defined as the amount of memory a process requires in a given time interval, the working set hints at the cache miss rate in a program. A larger working set would require a larger cache size to exploit temporal locality.
- Data Usage: High data sharing indicates that different processes are using the same sets of data, and as such rely more on the coherence protocols to keep the data synchronised. Data exchange offers a measure of this degree of sharing specific to bus traffic.

Bodytrack experiences the highest amount of data sharing and exchange.

Blackscholes is the smallest benchmark of the three, also having the least amount of data sharing. The working set is small.

Fluidanimate has the largest working set.

4.2 Methodology

We ran three different experiments for all implemented protocols, collecting performance statistics while independently varying the three simulator parameters as stated below. The base configuration is a 4KB 2-way set associative cache with a 32-byte block size.

- Block Size
- Associativity
- Cache Size

[3] "The PARSEC Benchmark Suite - ResearchGate." https://parsec.cs.princeton.edu/doc/parsec-report.pdf. Accessed 12 Nov. 2020.

For each of the experiments, the following statistics were collected:

- Overall Execution Cycle
- Average Cache Miss Rate
- Average Shared Data Access
- Invalidations/Updates
- Bus Traffic

4.3 Results

The full set of results collected for the three experiments can be found in the Appendix. In this section, we walk through some observations and our analysis of them.

Block Size

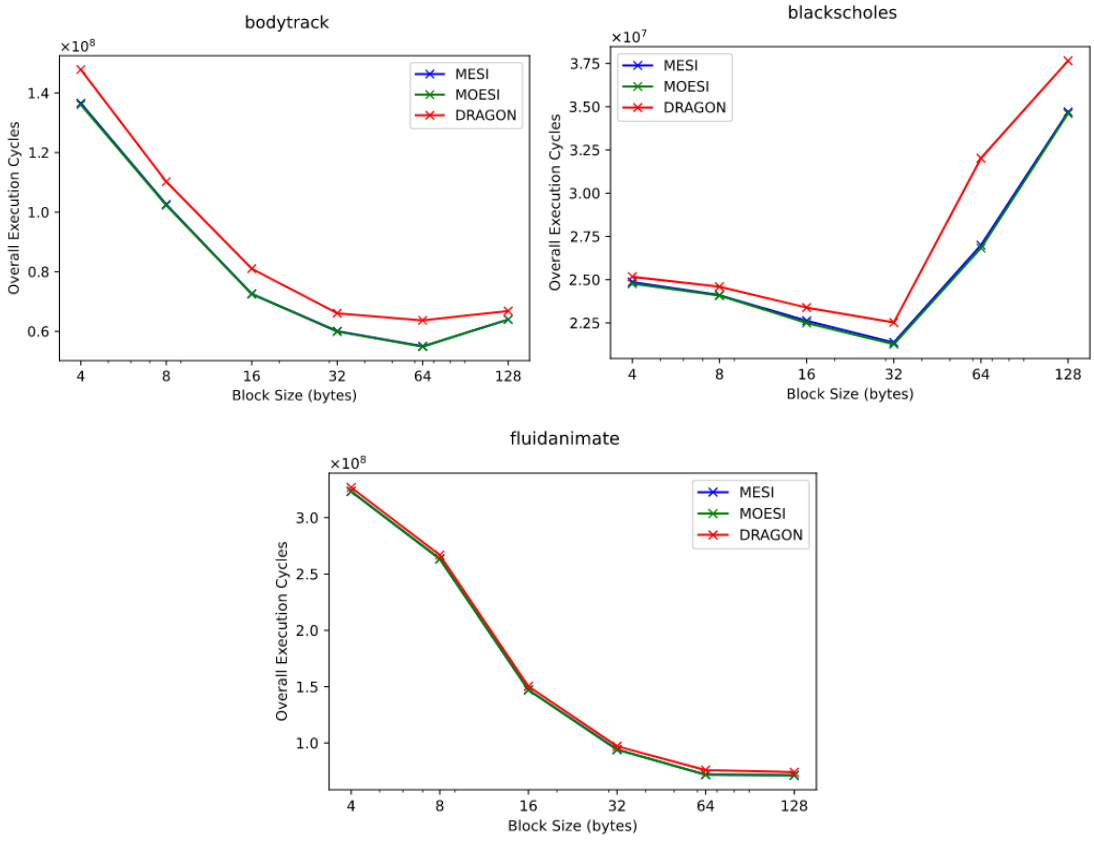

Figure 5. Overall Execution Cycle against Block Size (bytes)

General Trend

In general, execution cycles first decrease as block size is increased, then plateau and even increase (with bodytrack and blackscholes). This is in line with the understanding that as block size increases, performance initially improves as more blocks are brought in with a cache line, exploiting spatial locality to reduce cold misses. However, given a fixed cache size and associativity, increasing block size also means decreasing the number of sets. This violates temporal locality, as fewer blocks with different tags can be persisted in the cache.

Eventually, this outweighs the benefits derived from spatial locality and causes execution time to increase.

Benchmark Comparison

We note that for the blackscholes benchmark experiences the trend reversal most drastically and at the smallest block size among the benchmarks. This is likely because the benchmark has the smallest working set and the least amount of data sharing, and as such would benefit greatly from temporal locality rather than spatial locality. At smaller block sizes, the cache is better able to persist the specific blocks in the working set and perform evictions infrequently. Conversely, fluidanimate does not experience the reversal within the bounds of block sizes used. The large working set would frequently cause evictions and is unlikely to significantly benefit from temporal locality, instead fully exploiting the spatial locality granted by larger block sizes.

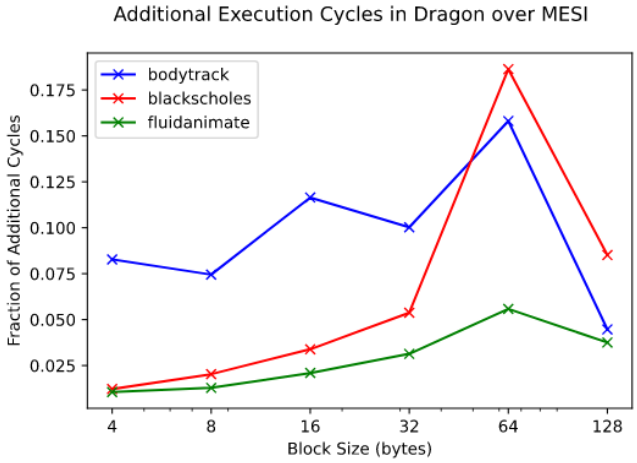

Figure 6. Fraction of Additional Execution Cycles against Block Size (bytes). Fraction of Additional Execution Cycles = $(C_{Dragon} - C_{MESI})/C_{MESI}$ where C_X is the overall execution cycle of protocol X

MESI vs Dragon

Between MESI and Dragon, it is observed that Dragon significantly underperforms in both the bodytrack and blackscholes benchmark, and slightly underperforms in the fluidanimate benchmark. We believe the reason invalidation performs significantly better than updates in bodytrack and blackscholes is that these benchmarks have coarser parallelization than fluidanimate, and thus have more sequentially shared data (rather than interleaved accesses from different processes). It has been previously shown that invalidation-based protocols perform better in these circumstances[4].

[4] "An Evaluation of Snoop-Based Cache Coherence Protocols." https://hps.ece.utexas.edu/people/suleman/class_projects/pca_report.pdf. Accessed 13 Nov. 2020.

Associativity

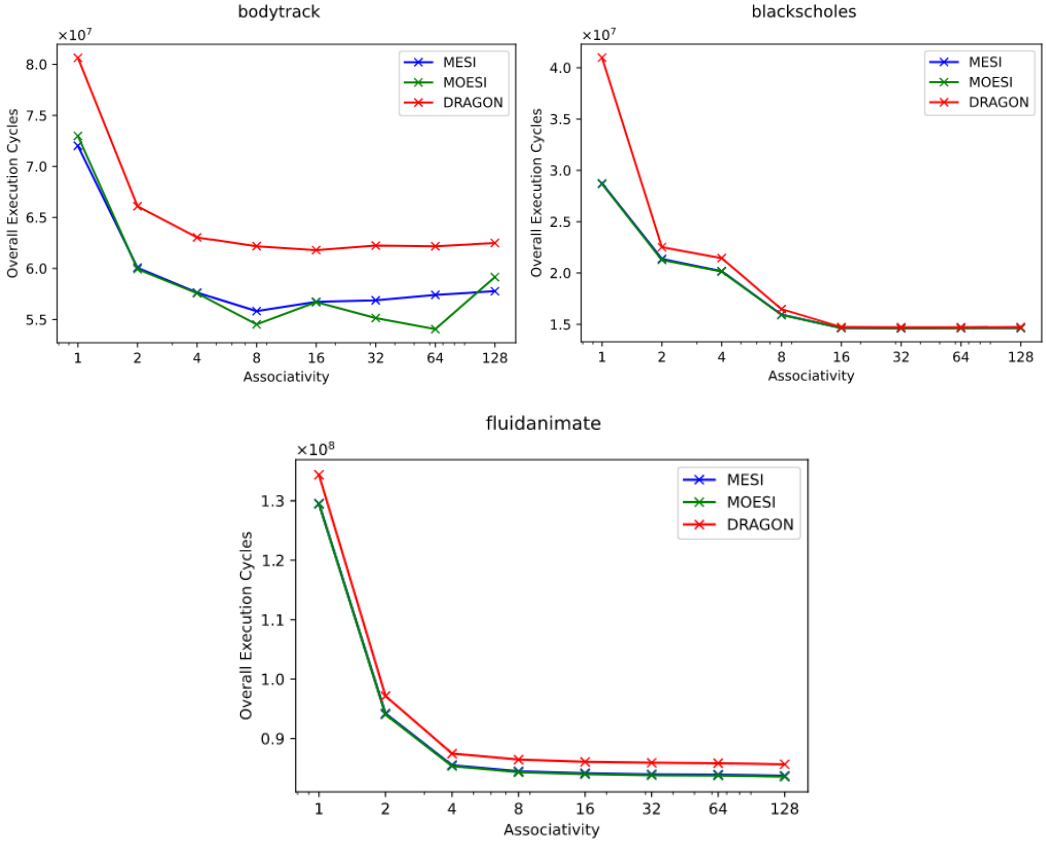

Figure 7. Overall Execution Cycle against Associativity

General Trend

The general trend observed is that higher associativities generally reduce overall execution time. A direct-mapped cache performs significantly poorer than a 2-way set associative cache, as it has poor temporal locality. As associativity increases up until the point of a fully associative cache (128), the overall execution cycle plateaus but does not increase (except for the bodytrack benchmark). One possible reason for this trend is that the additional amount of time required to search all cache blocks in a set is not modeled in the simulator, and thus direct mapping does not pose any drawbacks to execution time. In reality, this would incur a greater time penalty on a cache hit.

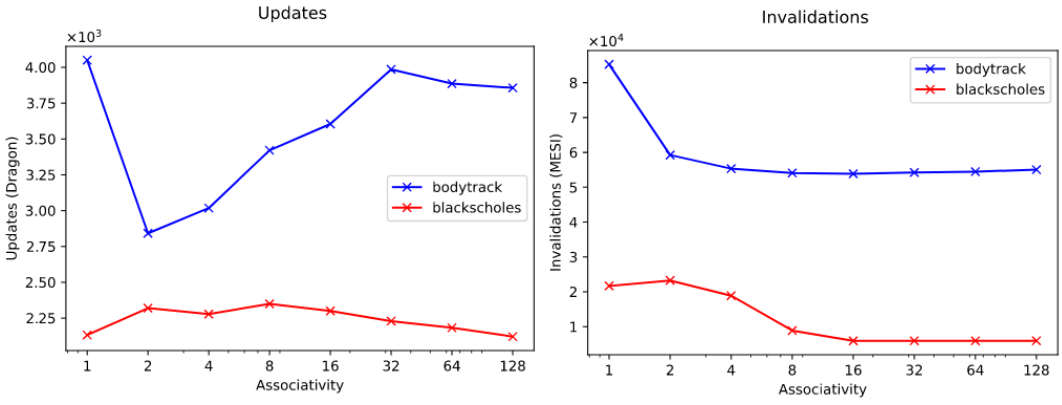

Figure 8. Updates (Dragon) & Invalidations (MESI) against Associativity

Benchmark Comparison

In Figure 7, the fluidanimate benchmark appears to begin experiencing diminishing returns at the lowest associativity (4). This is likely because a large working set would be least able to benefit from the temporal locality offered by larger cache sets. Another interesting observation is that the bodytrack benchmark begins to see a performance reduction after going beyond 8-way associativity. We believe that the greater temporal locality of higher associativities, coupled with the high data sharing of the benchmark, cause a greater number of invalidations/updates. This is verified in Figure 8 (for invalidations, the trend is more apparent in the raw data). While the blackscholes benchmark sees falling updates and invalidations as it approaches full associativity, the bodytrack benchmark begins seeing a rise in updates and invalidations beyond associativity values 2 and 16 respectively. The increase in bus traffic caused by these operations could be a factor leading to the reduced performance at higher associativities in the bodytrack benchmark.

MESI vs Dragon

Between MESI and Dragon, we once again observe that Dragon underperforms in all benchmarks, particularly in bodytrack and blackscholes. We attribute it to the same reason highlighted in the block size experiment.

Cache Size

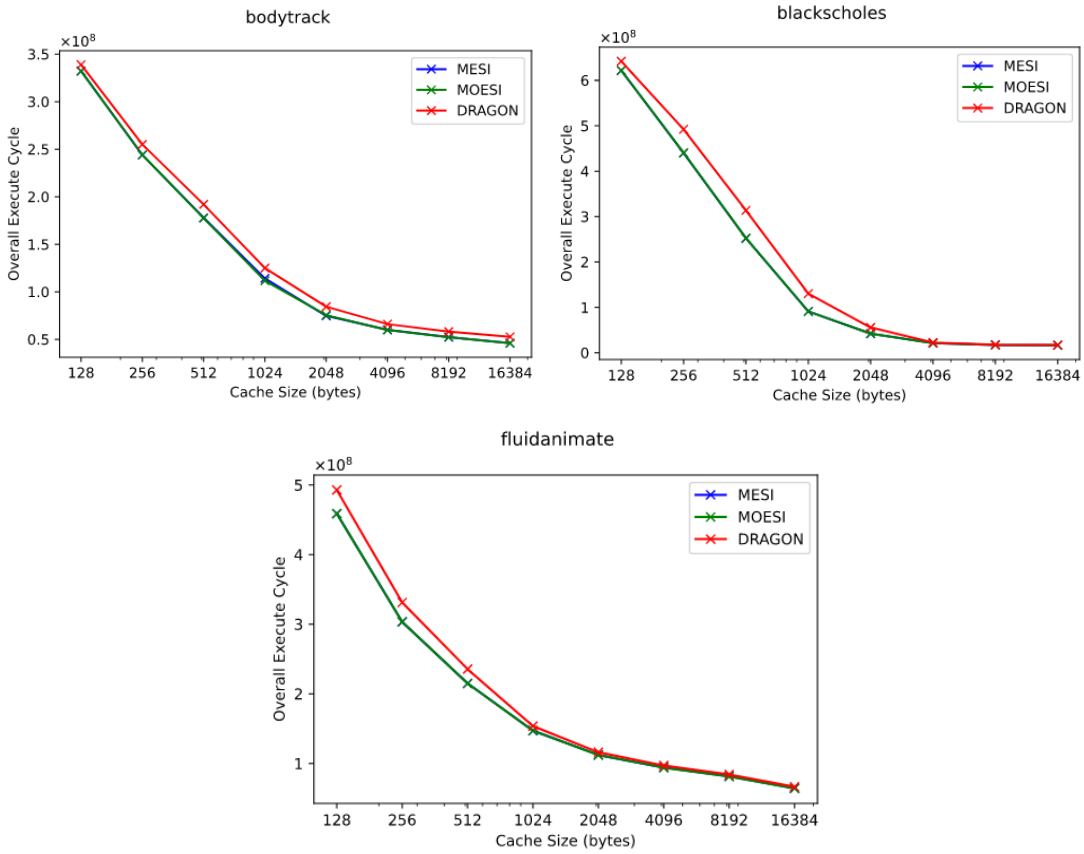

Figure 9. Overall Execution Cycle against Cache Size (bytes)

General Trend

The general trend observed when cache size is increased is that the overall execution cycles are reduced. This is likely because a larger cache size greatly improves temporal locality, leading to fewer cache misses. The downward curve plateaus as increasing cache size produces diminishing returns, as most of the addresses already do not face conflict misses and are unable to reap the benefits of even greater temporal locality.

Benchmark Comparison

The fluidanimate benchmark benefits the most from increased cache size, reaping reductions in performance cycles up until 16KB. On the other hand, blackscholes plateaus around 4KB. We believe that this can be attributed to the difference in the size of the working set; the small working set of the latter benchmark would have few conflict misses and be able to fully contain all data with less cache memory.

MESI vs Dragon

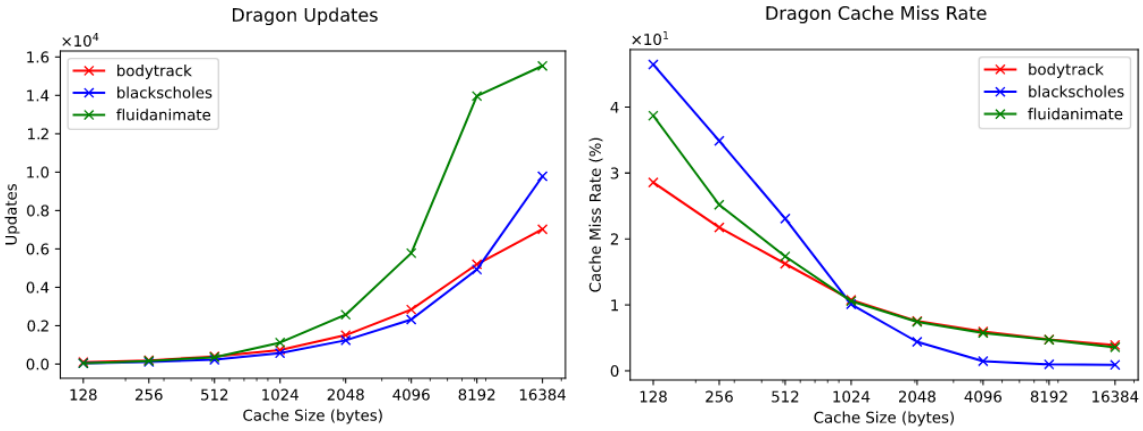

Figure 10. Dragon Updates and Cache Miss Rate (%) against Cache Size (bytes)

Both MESI and Dragon exhibit the downward trend as described. It is worth noting that for update-based protocols like Dragon, a larger cache size incurs greater bus overhead as blocks are kept in the cache for a longer duration, and will have to be updated. This will take place even if the block is not recently used, and raises the overall number of updates as seen in the left subplot of Figure 10. However, this is still outweighed by the reduced number of cache misses (right subplot in Figure 10), leading to the overall improvement in performance observed in Figure 9.

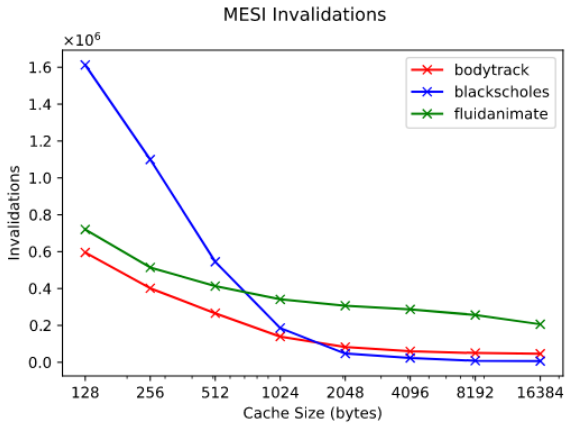

Figure 11. MESI Invalidations against Cache Size (bytes)

Unlike update-based protocols, invalidation-based protocols do not face drawbacks as cache size is increased. Greater temporal locality of cache blocks might potentially lead to more invalidation of data in S or E states, but such invalidations only incur small amounts of bus traffic (as no data is transferred). In fact, we observe that the number of invalidations decrease as cache size is increased. As there are more cache sets, there are fewer occurrences of M state blocks being evicted because the set is full. Thus, a later write to the block would not invoke a BusRdX transaction and invalidation. This was verified by logging the number of BusRdX that occurs, which we have done in an additional experiment (result shown in Appendix).

Advanced Task: MOESI

In our experiments, we observe largely similar performance between MESI and MOESI across benchmarks. In most configurations, MOESI outperforms MESI slightly. This implies that the additional O state introduced is successfully reducing the number of writebacks performed to memory and minimising the delays associated with the operation.

However, in our associativity and cache size experiments, there are a few specific configurations where MOESI performs poorer than MESI. As the only divergence between the execution of the two protocols takes place when an M state block is read on the bus (MESI: transit to I, MOESI: transit to O), we speculate that the reason MOESI underperforms is that delaying the write back to memory might generate more bus traffic:

- In MESI, when a M state block is read, it is written back to memory while supplying the requesting cache with data. The operations are overlapped and the number of bytes transmitted is the size of a single block. The M state block transits to S and can be evicted later (due to cache reaching capacity) without incurring a write back.
- In MOESI, the data is sent over to the requesting cache but not written back to memory. The M state block transits to the O state. If the O state block is evicted later, an additional bus transaction carrying a block is incurred for the write back to memory.

Thus, if the above scenario happens frequently, the benefit of keeping dirty data in the cache without immediately writing back is outweighed.

While the performance differences between the two protocols was mostly negligible, it was more apparent in our associativity experiment (Figure 7), particularly for the bodytrack benchmark. We speculate that the high data sharing in the benchmark produces more scenarios where blocks enter the O state, amplifying the differences in MESI and MOESI.

5 Conclusion

Across all benchmarks and cache configurations tested, MESI consistently performs better than Dragon. This is even in a circumstance where update-based protocols are known to outperform invalidation-based protocols, namely when there is fine-grained parallelization and interleaved reads/writes by multiple processors (i.e. the fluidanimate benchmark). Two possible reasons for this observation are:

1. The benchmarks provided do not interleave accesses to a degree for Dragon to perform better than MESI
2. Our implementation assumptions grant advantages to MESI that close the gap in performance with Dragon. One such assumption could be having the bus pick one responder to reply to a read in MESI (thus avoiding bombarding the bus with responses), which may not be true depending on the actual MESI implementation.

Given our assumptions and findings, we conclude that MESI is always preferable to Dragon for the bodytrack, blackscholes and fluidanimate PARSEC benchmarks in terms of execution performance.

We have also found that MOESI performs marginally better compared to MESI across all benchmarks, with the exception of a few cache configurations. We thus find MOESI a meaningful optimisation of the MESI protocol in general.

6 Appendix

6.1 Block Size

MESI

bodytrack					
Block Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
4	136555758	12.36	22.73	323482	5909852
8	102588908	9.67	18.80	197329	8766120
16	72595569	7.27	22.64	103418	12776176
32	60074386	5.96	21.27	59235	20584928
64	54997270	5.78	28.42	42701	39446720
128	63981692	6.38	18.02	35963	82633728

blackscholes					
Block Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
4	24867011	1.52	19.22	40314	776112
8	24107494	1.45	19.05	37186	1471696
16	22623887	1.46	18.95	27913	2919056
32	21372539	1.47	18.82	23265	5715264
64	26995153	2.50	17.99	20584	19260416
128	34698211	3.02	17.43	30415	46322432

fluidanimate					
Block Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
4	323380385	19.13	18.62	1163647	12822352
8	263472312	15.01	17.04	977927	20792848
16	147205239	8.60	16.36	510071	23143040
32	94202851	5.75	16.71	287157	29566848
64	71955495	4.84	17.01	175939	46043136
128	71398247	5.09	17.50	127595	90857600

MOESI

bodytrack					
Block Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**

4	136090525	12.36	23.43	323520	5910320
8	102309058	9.67	19.00	197342	8766672
16	72500297	7.27	22.62	103417	12777616
32	59933636	5.96	21.45	59182	20587904
64	54860420	5.78	28.51	42732	39480896
128	63980092	6.38	17.87	35954	82651776

blackscholes					
Block Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
4	24768650	1.52	19.21	40311	776236
8	24085242	1.45	19.05	37185	1471928
16	22500745	1.46	18.95	27914	2919376
32	21270448	1.47	18.82	23265	5716032
64	26822835	2.50	17.99	20598	19262848
128	34607071	3.02	17.45	30421	46328064

fluidanimate					
Block Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
4	323241653	19.13	18.61	1163639	12822504
8	263320517	15.01	17.04	977916	20793024
16	147051501	8.60	16.36	510072	23143360
32	94063371	5.75	16.68	287147	29569184
64	71831390	4.84	17.01	175961	46052544
128	71333417	5.09	17.50	127647	90877312

Dragon

bodytrack					
Block Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Updates**	**Bus Traffic (bytes)**
4	147856592	12.34	18.91	4672	5919604
8	110230735	9.65	17.22	4635	8784640
16	81049859	7.25	15.50	3802	12814288
32	66097456	5.95	15.53	2842	20652320
64	63691344	5.76	16.18	3194	39611136
128	66841274	6.37	18.08	2726	82963968

blackscholes					
Block Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg.**	**Updates**	**Bus Traffic (bytes)**

			(%)		
4	25170322	1.52	19.32	6142	799032
8	24596816	1.44	19.17	6463	1520072
16	23389632	1.46	19.02	3688	2974512
32	22521833	1.47	18.86	2320	5785728
64	32024716	2.50	18.01	1565	19354944
128	37651675	3.02	17.52	1145	46457728

fluidanimate					
Block Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Updates**	**Bus Traffic (bytes)**
4	326795868	19.13	18.64	4902	12840968
8	266850712	15.01	16.99	3622	20820800
16	150290204	8.59	16.40	4777	23213728
32	97155486	5.74	16.69	5779	29746976
64	75973164	4.84	17.21	6229	46402496
128	74077047	5.09	17.50	3016	91174912

6.2 Associativity

MESI

bodytrack					
Associativity	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
1	72019985	7.19	19.92	85255	25313728
2	60074386	5.96	21.27	59235	20584928
4	57641839	5.58	22.09	55328	19508704
8	55822257	5.49	23.96	54102	19166176
16	56724616	5.48	22.38	53868	19130944
32	56882664	5.45	22.16	54241	19185792
64	57412513	5.47	21.39	54480	19279424
128	57774295	5.49	20.61	55044	19381216

blackscholes					
Associativity	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
1	28729821	3.66	16.55	21691	12687232
2	21372539	1.47	18.82	23265	5715264
4	20177722	1.39	18.79	18875	5268544
8	15940220	0.65	18.80	8883	2415104

16	14648613	0.25	18.87	5934	1025696
32	14620340	0.25	18.87	5949	1015264
64	14638100	0.25	18.87	5942	1015520
128	14640431	0.25	18.87	5958	1017344

fluidanimate					
Associativity	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
1	129541063	8.68	16.99	359781	42218560
2	94202851	5.75	16.71	287157	29566848
4	85524355	4.90	17.04	271733	26054752
8	84493981	4.76	17.15	272895	25540256
16	84168682	4.72	17.13	273177	25350144
32	83992710	4.69	17.29	273265	25262080
64	83923586	4.68	17.20	273450	25227456
128	83744288	4.65	17.26	273550	25112000

MOESI

bodytrack					
Associativity	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
1	72984425	7.18	16.70	85268	25315872
2	59933636	5.96	21.45	59182	20587904
4	57569059	5.58	21.94	55256	19514144
8	54519631	5.49	26.01	54206	19170720
16	56701338	5.48	22.36	53887	19136576
32	55147094	5.45	25.11	54398	19192768
64	54058158	5.48	27.22	54741	19281824
128	59155759	5.49	14.43	54957	19386880

blackscholes					
Associativity	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
1	28671566	3.66	16.54	21688	12687712
2	21270448	1.47	18.82	23265	5716032
4	20141986	1.39	18.79	18871	5269056
8	15916234	0.65	18.80	8883	2415776
16	14645527	0.25	18.87	5930	1026048
32	14626923	0.25	18.87	5948	1015808
64	14632015	0.25	18.87	5945	1016224

128	14634513	0.25	18.87	5955	1018080

fluidanimate					
Associativity	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
1	129452217	8.68	16.97	359794	42221120
2	94063371	5.75	16.68	287147	29569184
4	85383958	4.90	17.05	271675	26056928
8	84340282	4.76	17.17	272903	25542368
16	84015886	4.72	17.12	273218	25353536
32	83838178	4.69	17.25	273313	25264832
64	83800665	4.68	17.22	273395	25232320
128	83588799	4.65	17.23	273517	25111872

Dragon

bodytrack					
Associativity	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Updates**	**Bus Traffic (bytes)**
1	80639052	7.18	14.46	4050	25424768
2	66097456	5.95	15.53	2842	20652320
4	63040730	5.57	16.63	3018	19586720
8	62186566	5.48	18.48	3421	19258944
16	61802398	5.48	12.39	3605	19235712
32	62246727	5.45	18.37	3986	19309632
64	62172761	5.47	11.00	3886	19408480
128	62496847	5.49	11.03	3857	19509632

blackscholes					
Associativity	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Updates**	**Bus Traffic (bytes)**
1	40985263	3.66	16.56	2132	12749376
2	22521833	1.47	18.86	2320	5785728
4	21453765	1.39	18.83	2278	5337216
8	16478969	0.65	18.84	2350	2486624
16	14733923	0.25	18.91	2300	1095072
32	14719778	0.24	18.91	2229	1082368
64	14727490	0.25	18.91	2183	1081664
128	14737310	0.25	18.91	2121	1080928

fluidanimate

Associativity	Total Execution Cycle	Cache Miss Rate Avg. (%)	Shared Data Access Avg. (%)	Updates	Bus Traffic (bytes)
1	134375791	8.67	17.0z	6069	42394048
2	97155486	5.74	16.69	5779	29746976
4	87475680	4.90	17.22	6704	26262592
8	86459684	4.76	17.26	5102	25706464
16	86078412	4.71	17.23	5033	25510080
32	85927701	4.69	17.32	4778	25416352
64	85860117	4.68	17.34	4912	25385152
128	85680070	4.65	17.33	4863	25264224

6.3 Cache Size

MESI

bodytrack					
Cache Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
128	332292735	28.57	3.28	595506	108309856
256	244005191	21.75	5.28	401239	81416736
512	177908900	16.24	8.42	265750	61230048
1024	114126906	10.77	15.97	139446	39769920
2048	74776967	7.54	24.35	82280	26736096
4096	60074386	5.96	21.26	59235	20584928
8192	52356287	4.77	22.72	50301	17641024
16384	46072763	3.93	29.80	46048	15891968

blackscholes					
Cache Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
128	621586522	46.46	1.59	1612426	205429888
256	440027050	34.89	5.63	1099847	157490912
512	252183475	23.09	8.21	544981	100349216
1024	91023890	10.10	13.28	185859	41592384
2048	42186667	4.40	16.79	47318	17530688
4096	21372539	1.47	18.82	23265	5715264
8192	17366670	0.97	19.05	8695	3396160
16384	16658282	0.90	19.10	6699	3091584

fluidanimate					
Cache Size	**Total Execution**	**Cache Miss**	**Shared Data**	**Invalidations**	**Bus Traffic**

(bytes)	Cycle	Rate Avg. (%)	Access Avg. (%)		(bytes)
128	458776462	38.71	7.35	720639	157648640
256	303311169	25.19	12.15	514163	105909376
512	215024534	17.36	14.18	413146	75219872
1024	147131722	10.57	16.00	341232	48780576
2048	112385832	7.42	16.28	306817	36284960
4096	94202851	5.75	16.71	287157	29566848
8192	81679005	4.72	17.76	256760	24964928
16384	64539538	3.58	18.82	206302	19034816

MOESI

bodytrack					
Cache Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
128	332135537	28.57	3.26	595504	108310688
256	244099250	21.75	5.37	401257	81417792
512	177576749	16.24	9.50	265726	61231616
1024	111713339	10.76	21.09	139480	39772384
2048	75501891	7.54	23.15	82320	26742848
4096	59933636	5.96	21.45	59182	20587904
8192	52243973	4.77	22.75	50349	17646144
16384	45955348	3.93	29.77	46033	15897472

blackscholes					
Cache Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
128	621924488	46.46	1.51	1612424	205429984
256	440335016	34.89	5.59	1099852	157491264
512	252276261	23.09	8.20	544991	100349792
1024	91077985	10.10	13.27	185871	41592928
2048	42236840	4.40	16.80	47320	17530976
4096	21270448	1.47	18.82	23265	5716032
8192	17365316	0.97	19.05	8695	3397120
16384	16629219	0.90	19.10	6702	3092192

fluidanimate					
Cache Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Invalidations**	**Bus Traffic (bytes)**
128	458745522	38.71	7.35	720634	157648768
256	303336035	25.19	12.14	514148	105910656

512	215017574	17.36	14.15	413137	75221216
1024	147065217	10.57	15.98	341249	48780704
2048	112315876	7.42	16.30	306807	36286784
4096	94063371	5.75	16.68	287147	29569184
8192	81502342	4.72	17.73	256810	24970016
16384	64318634	3.58	18.81	206300	19038560

Dragon

bodytrack					
Cache Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Updates**	**Bus Traffic (bytes)**
128	338974212	28.57	3.34	105	108313312
256	255076496	21.75	9.06	195	81421952
512	192036762	16.24	7.98	400	61237536
1024	125027619	10.76	13.72	737	39788512
2048	84437889	7.54	15.12	1506	26773344
4096	66097456	5.95	15.53	2842	20652320
8192	58118138	4.75	23.33	5198	17763232
16384	52716291	3.91	19.81	7025	16061408

blackscholes					
Cache Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Updates**	**Bus Traffic (bytes)**
128	641991113	46.46	1.05	42	205431072
256	492202170	34.89	2.38	114	157494048
512	313686404	23.09	6.59	226	100355136
1024	130275649	10.10	13.25	569	41608256
2048	55708848	4.40	16.83	1239	17567040
4096	22521833	1.47	18.86	2320	5785728
8192	17769791	0.97	19.14	4927	3549568
16384	16960950	0.90	19.27	9782	3396704

fluidanimate					
Cache Size (bytes)	**Total Execution Cycle**	**Cache Miss Rate Avg. (%)**	**Shared Data Access Avg. (%)**	**Updates**	**Bus Traffic (bytes)**
128	492701734	38.71	7.28	43	157649920
256	331288913	25.19	12.16	166	105914016
512	235560070	17.36	14.16	356	75230208
1024	153763900	10.57	15.97	1118	48813152
2048	116180184	7.42	16.21	2575	36361632

4096	97155486	5.74	16.69	5779	29746976
8192	84389900	4.71	17.84	13964	25383968
16384	66627373	3.57	18.66	15535	19543840

BusRdX vs BusUpgr

Additional experiment to verify that BusRdX decreases at a rate faster than BusUpgr increases as cache size increases.

MESI

bodytrack			
Cache Size (bytes)	**BusRdX**	**BusUpgr**	**Invalidations**
128	595441	65	595506
256	401155	84	401239
512	265591	159	265750
1024	139107	339	139446
2048	81412	868	82280
4096	58008	1227	59235
8192	48774	1527	50301
16384	43962	2086	46048