CS4223 Assignment 2 Report

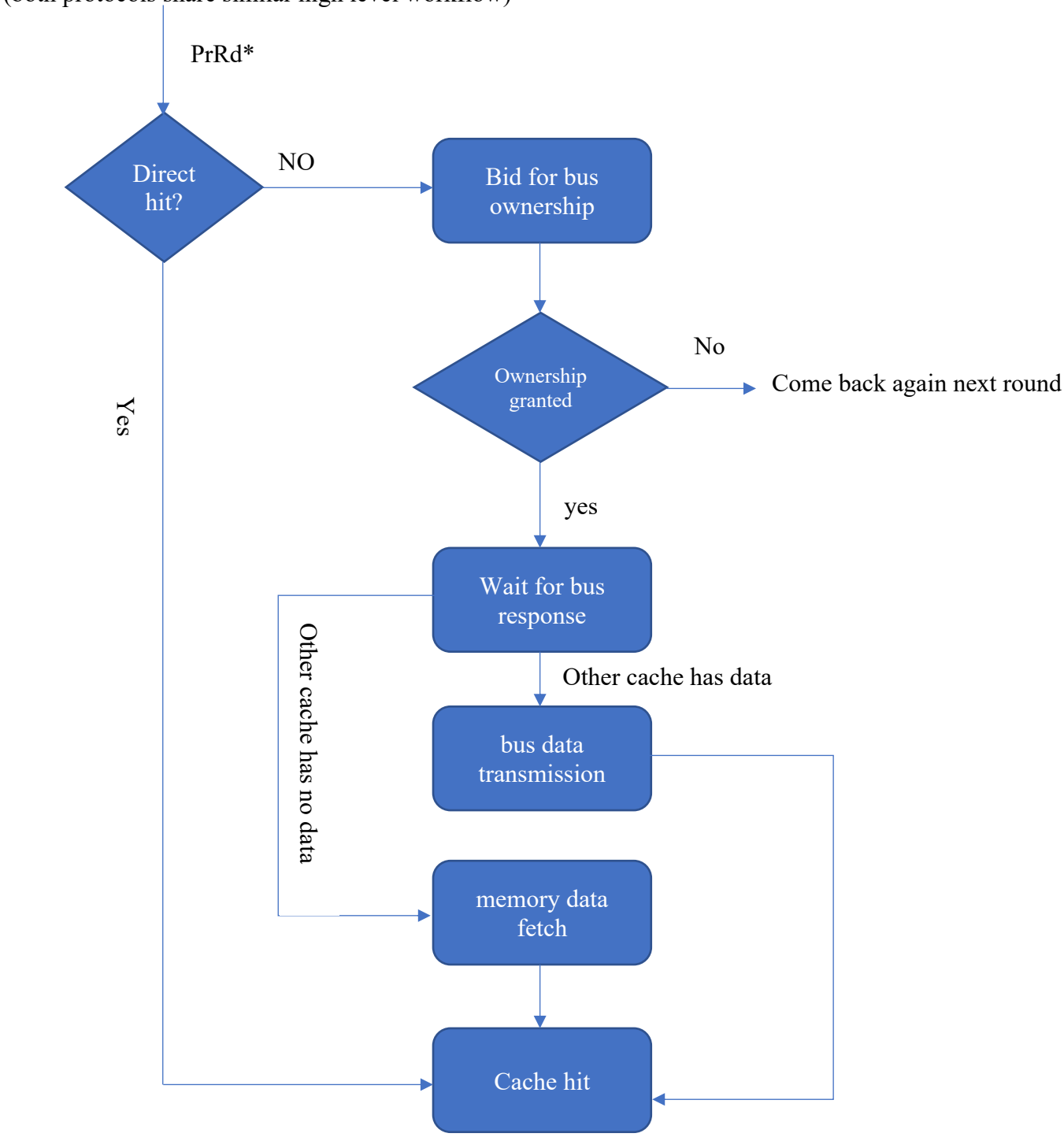
Additional Assumptions

- Memory Controller
    1. Memory can handle infinite concurrent fetches and each fetch takes 100 cycles sharp.
- Bus
    1. At any point of time, only one cache can own the bus. Ownership can only be released by the owner itself.
    2. Bidding and the subsequent granting of bus ownership does not take time, it happens at the instant right before a clock cycle starts, in other words it happens during the *interim* between two clock cycles. Bidding is only allowed if the bus currently has no owner.
    3. Bus ownership can only be granted to one cache once per *interim*. Once a cache becomes the bus owner, other bidders have to back off and wait till the beginning of next cycle to come back and check the availability again, even though the bus ownership may immediately be released by its owner (e.g. a BusRdX that only causes invalidation but no data transfer)
- Locking and Atomicity
    1. In addition to the cache line states, we introduce a 3-state locking bit for each cache line: **unlocked**, **read_locked**, **write_locked**.
    2. A cache block in **unlocked** state is subject to reads and invalidations by all incoming bus requests.
    3. A **read_locked** block can respond to BusRd, but BusRdX or BusUpd requests will stall and only get their response after the lock is released.
    4. A **write_locked** block will stall all incoming bus requests that is targeting that block, only responding to them after the lock is released.
    5. During a read hit, the cache block will be **read_locked**. During a write hit it shall be **write_locked**.
    6. When fetching a block from memory, the block being fetched will be **write_locked**.
    7. When fetching a block from other caches thru the bus, the block will be **write_locked**.
    8. When a bus request for a block does not return positive answers (meaning the block is not found in any of the other caches), the bus ownership will be released. The cache will proceed to fetch the block from memory. Although the bus can now be grabbed by other caches and other cache may want to read this block, due to the aforementioned write lock still in place, access to this block is forbidden thus retaining atomicity for the entire operation.
- Block replacement
    1. When a processor tries to access an address that is not to be found in the corresponding cache set, the eviction of the least recently accessed block **will not happen immediately**. The eviction will only commence when the cache successfully secures the ownership of the bus. If the bus ownership is in other cache's hands, the block will stay for now so whoever owns the bus may still send a BusRd/BusRdX to access/invalidate it before it gets evicted.

2. If any cache access results in a block eviction, the cache will be stalled, waiting for the eviction to complete before proceeding with either bus transaction or memory fetch, effectively adding 100 cycles to the operation.
3. In the case where local Modified block gets flushed by incoming BusRdX, we assume such passive eviction runs synchronously, i.e. eviction will stall the cache.
4. There is no memory bandwidth limit for each core, i.e. multiple memory transactions can be run simultaneously.

Implementation Flowchart

(both protocols share similar high level workflow)



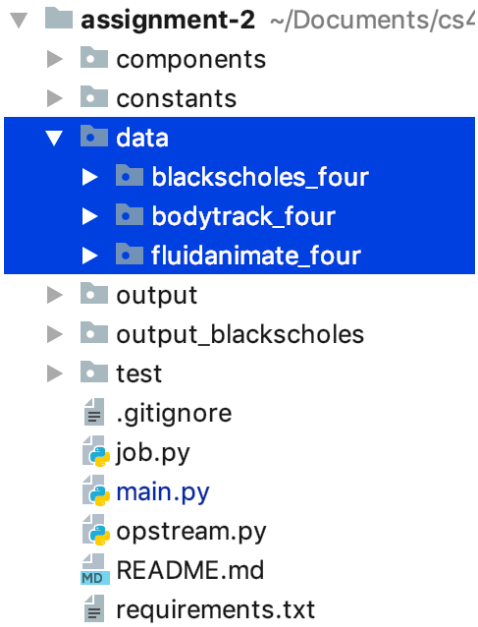
* Flow chart for PrWt is very similar, only that from *wait for bus* you can go directly to *Cache Hit*, if local cache already has data in Shared state.

Key Data Structure

- Cache Data Representation
  - A 3-dimensional numpy array of shape (m, n, 3)
  - m is the number of cache sets given by cache_size/associativity/block_size
  - n is the associativity
  - last dimension represents 3-element tuple of [tag, state, lockbit]
- Processor Class
  - In charge of issuing new instruction if available.
  - Counting for non-cache instructions.
- Cache Class
  - Maintains the simulated cache data.
  - Implementation of different state transition mechanisms provided by different protocol.
- Bus Class
  - Connect each cache class.
  - Support bus-initiated data transistions.

Running the Simulator

- Language
  - Python 3
- Setup
  - pip install -r requirements.txt
- folder structure
  - make sure the data folder is put under the root directory of the source code.



- Run, e.g.
  - ./coherence.sh MESI ./data/bodytrack_four 1024 1 16
- Sample Output

```
Run:  main
All Finished! Current counter: 39703006
processsor 0 counter: 39272184
processsor 1 counter: 39703006
processsor 2 counter: 18846743
processsor 3 counter: 39283717
                                stats
Bus Data Traffic                5292992
Bus Invalidation/Updates          59541
Overall Execution Cycle         39703006
Private Data Access Percentage        69
                                    P0           P1           P2           P3
Compute Cycles              17729254.00  17120545.00  17556877.00  17140113.00
Load/Store Instructions      3270132.00   3287252.00    117698.00   3324919.00
Idle Cycles                 21542930.00  22582461.00   1289866.00  22143604.00
Cache Miss Rate                    0.03         0.04         0.04         0.03

Process finished with exit code 0
```

Advanced Task

Improvement to MESI

MESI-based protocol is more commonly used in modern multi-core architecture. For example, Intel used MESIF for its Core Series CPU. Among many variants of MESI-based protocols, we select MOESI as our additional coherence protocol.

Compared with MESI, MOESI simply add an OWNED state to the MOESI to reduce redundant memory accesses. When a cache line in M state, and a BusRd ocurrs, this cache line will transition to O state, rather than writing back to memory. O state is similar to S state, except that cache in O state have to write back to main memory, when this cache is to be replaced. The following figure shows the FSM of MOESI.

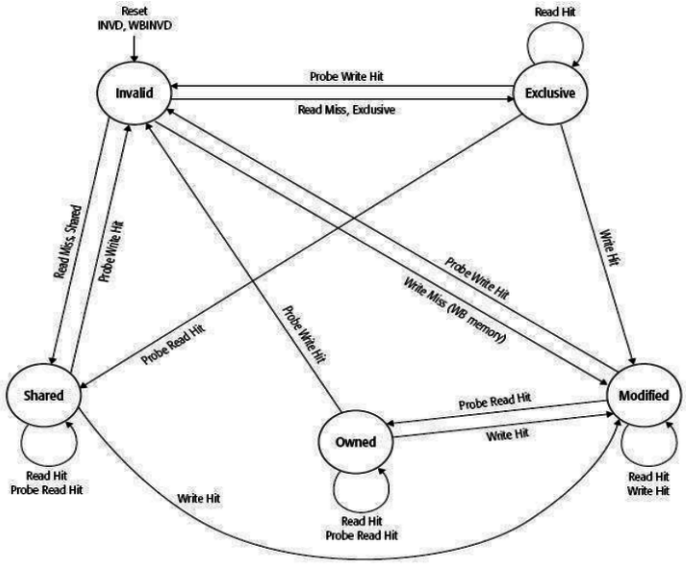


Fig source: Dey, Somdip, and Mamatha S. Nair. "Design and Implementation of a Simple Cache Simulator in Java to Investigate MESI and MOESI Coherency Protocols." *International Journal of Computer Applications* 87.11 (2014).

Improvement to Dragon

For the Dragon protocol, there are some redundant memory operations shown in many state transition graphs. Just like the figure below, if a cache block is in Sm or M state, and a BusRd occurs, then the cache block will be flush to the main memory. Such operation is actually redundant if we consider memory write-back every time caches with Sm or M state is

replaced. In this way, our improvement is to remove the memory flush in Sm and M state. We name this modified protocol Dragon-noflush

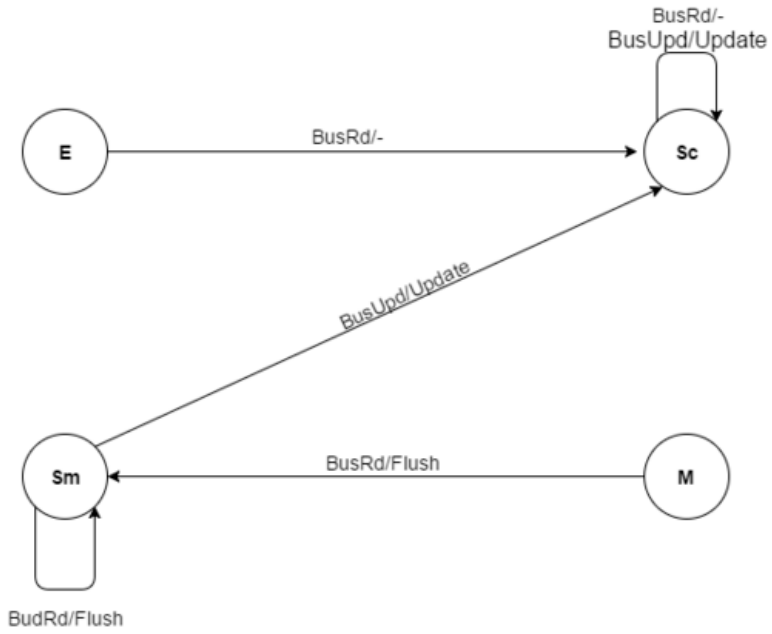


Fig source: wikipedia

Result Analysis

(Configuration: 4KB 2-way associative cache, block size 32 bytes, word size 4 bytes)

1. Black Scholes

	MESI			
	P0	P1	P2	P3
Compute Cycles	10430314	10383276	10430338	10394904
Load/Store Instructions	2497349	2490468	2509057	2503127
Idle Cycles	6636123	6589273	7549819	6726709
Cache Miss Rate	0.01	0.01	0.01	0.01
Bus Data Traffic (Byte)	697216			
Bus Invalidation/Updates	23290			
Overall Execution Cycle	17980157			
Private Data Access Percentage	81			
	DRAGON			
	P0	P1	P2	P3
Idle Cycles	6640924	6610163	7556377	6734058
Cache Miss Rate	0.01	0.01	0.01	0.01
Bus Data Traffic (Byte)	715456			
Bus Invalidation/Updates	26496			
Overall Execution Cycle	17986715			
Private Data Access Percentage	81			

	MOESI			
	P0	P1	P2	P3
Idle Cycles	6635323	6575998	7547403	6725986
Cache Miss Rate	0.01	0.01	0.01	0.01
Bus Data Traffic (Byte)	693408			
Bus Invalidation/Updates	23293			
Overall Execution Cycle	17977741			
Private Data Access Percentage	81			
	DRAGON-noflush			
	P0	P1	P2	P3
Idle Cycles	6643822	6603519	7556814	6732024
Cache Miss Rate	0.01	0.01	0.01	0.01
Bus Data Traffic (Byte)	693408			
Bus Invalidation/Updates	23293			
Overall Execution Cycle	17987152			
Private Data Access Percentage	81			

2. Body Track

	MESI			
	P0	P1	P2	P3
Compute Cycles	17729254	17120545	17556877	17140113
Load/Store Instructions	3270132	3287252	117698	3324919
Idle Cycles	21819530	22859061	1299568	22420204
Cache Miss Rate	0.03	0.04	0.04	0.03
Bus Data Traffic (Byte)	4984160			
Bus Invalidation/Updates	59476			
Overall Execution Cycle	39979606			
Private Data Access Percentage	71			
	DRAGON			
	P0	P1	P2	P3
Idle Cycles	21675503	22715113	1288849	22276250
Cache Miss Rate	0.03	0.04	0.04	0.03
Bus Data Traffic (Byte)	5064828			
Bus Invalidation/Updates	60975			
Overall Execution Cycle	39835658			
Private Data Access Percentage	70			
	MOESI			
	P0	P1	P2	P3
Idle Cycles	21542930	22582461	1289866	22143604
Cache Miss Rate	0.03	0.04	0.04	0.03
Bus Data Traffic	5292992			
Bus Invalidation/Updates	59541			

Overall Execution Cycle	39703006			
Private Data Access Percentage	69			
	DRAGON-noflush			
	P0	P1	P2	P3
Idle Cycles	21645736	22685346	1296561	22246483
Cache Miss Rate	0.03	0.04	0.04	0.03
Bus Data Traffic	5053300			
Bus Invalidation/Updates	60880			
Overall Execution Cycle	39805891			
Private Data Access Percentage	71			

3. Fluid Animate

	MESI			
	P0	P1	P2	P3
Compute Cycles	11337782	11290799	11337671	11301515
Load/Store Instructions	2576503	2407844	2604189	2411465
Idle Cycles	33174662	31502225	37946761	38935559
Cache Miss Rate	0.03	0.02	0.03	0.02
Bus Data Traffic (Byte)	1168832			
Bus Invalidation/Updates	287074			
Overall Execution Cycle	50237074			
Private Data Access Percentage	84			
	DRAGON			
	P0	P1	P2	P3
Idle Cycles	33132852	31431073	37840315	38870638
Cache Miss Rate	0.03	0.02	0.03	0.02
Bus Data Traffic (Byte)	1119728			
Bus Invalidation/Updates	289340			
Overall Execution Cycle	50172153			
Private Data Access Percentage	85			
	MOESI			
	P0	P1	P2	P3
Idle Cycles	33077951	31380752	37798948	38787746
Cache Miss Rate	0.03	0.02	0.03	0.02
Bus Data Traffic (Byte)	1171584			
Bus Invalidation/Updates	287074			
Overall Execution Cycle	50089261			
Private Data Access Percentage	84			
	DRAGON-noflush			
	P0	P1	P2	P3
Idle Cycles	33055461	31350984	37780509	38747811

Cache Miss Rate	0.03	0.02	0.03	0.02
Bus Data Traffic (Byte)	1116296			
Bus Invalidation/Updates	289397			
Overall Execution Cycle	50049326			
Private Data Access Percentage	85			

From the testing results of 3 cache coherence protocols on different benchmark trace, we can see that different protocols perform differently upon the benchmark used. We try to make some analysis based on the results.

1. MESI and DRAGON perform quite similarly on 3 benchmarks under the same cache configuration. For the *blackscholes*, MESI has fewer overall execution cycles, while for *bodytrack* and *fluidanimate*, DRAGON outperforms MESI. To analyze, we know that different coherence protocols can cause different number of accesses to main memory, and different amount of data transferred via bus. Except for *fluidanimate* benchmark, DRAGON has more bus data traffic. By considering the differences between invalidation protocol and update protocol, more data write operations will cause more performance loss for update protocol, which also implies that our testing benchmarks generally have a considerable number of data write operations.
2. MOESI and DRAGON-noflush perform better than their corresponding naïve version in our testing benchmarks. Thanks to the OWNED state, overall execution cycles of MOESI are reduced in all testing benchmarks. For the DRAGON-noflush, the overall execution cycles for *bodytrack* and *fluidanimate* are reduced, while overall execution cycles for *blackscholes* are increased a bit, which may be caused by the different coordination of bus requests.

To conclude, the MOESI protocol has the best overall performance of all three testing benchmarks, which also shows the reason why modern CPU prefer to use MESI or MOESI alike protocols.