**CS4223 Assignment 2**

Assignment 2 revolves around implementing MESI Invalidation Protocols and Dragon Update Protocols in a quad-core system. I implement the above protocols in Python 3.7. Each protocol is then analysed using the below 3 sets of instructions provided:

1. fluidanimate
2. blackscholes
3. bodytrack

To build this simulation, I build 3 classes – Processor, Cache and Bus. Each of the 4 processors (P0 – P3) will have a 4KB 2-way set associative cache, with a 32-byte block size by default, as stated in the handout. Since the address has 32 bits as defined by the handout, the address format is shown below. 5 bits are needed for block offset are the block size is 25 bytes. The number of sets is equal to which will be 64 based on the default settings, which means the index is represented by 6 bits as 64 = 26. The rest of the address will be interpreted as the tag. The format of the address is shown in Figure 1 below.

10

11

31

5

0

4

|  |  |  |
| --- | --- | --- |
| Tag | Set Index | Block Offset |

Figure 1. Structure of address

All the 4 caches will be connected to the bus and will trigger bus transactions depending on the state of the cache line and instruction being executed. To implement both of those protocols, I created MESICache and DragonCache classes on top of the Cache class as different protocols affects how the cache will behave. In addition, the LRU functionality is implemented by the OrderedDict class in Python, which enables us to maintain sequential order of the blocks based on when they were used.

These are the classes implemented and a short explanation of what each method does:

1. Processor
   1. execute(curr\_cycle) – Read the instructions on the trace file line by line
   2. process(trace) – Execute the instructions read from trace file
   3. load(memory\_address) – Tries to load from a memory address
   4. store(memory\_address) – Tries to store in memory address
   5. report() – Print out statistics such as the execution cycles
2. Block
   1. Contains attributes like state, tag and words
3. Cache
   1. process\_address(memory\_address) – Convert the address in hexadecimal to bits and return the tag, set index and block offset
   2. in\_cache(tag, set\_index, offset) – Check for valid block at that particular set index having the same tag
   3. cache\_line\_full(tag, set\_index, offset) – Check of the cache line at the set index is full to determine whether the least recently used need to be evicted
4. MESICache
   1. load(memory\_address) – Loads data from a memory address
   2. store(memory\_address) – Stores data from a memory address
5. DragonCache
   1. load(memory\_address) – Loads data from a memory address
   2. store(memory\_address) – Stores data from a memory address
6. Bus
   1. mesi\_read(id, tag, set, offset) – BusRd for MESI Protocol
   2. mesi\_readx(id, tag, set, offset) – BusRdx for MESI Protocol
   3. mesi\_upgr(id, tag, set, offset) – BusUpgr for MESI Protocol
   4. mesi\_flush(id, tag, set, offset) – Flush for MESI Protocol
   5. dragon\_read(id, tag, set, offset) – BusRd for Dragon Protocol
   6. dragon\_update(id, tag, set, offset) – BusUpd for Dragon Protocol
   7. dragon\_flush(id, tag, set, offset) – Flush for Dragon Protocol

**1. MESI Protocol Statistics**

**1.1. Benchmark 1 - fluidanimate**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Processor 0 | Processor 1 | Processor 2 | Processor 3 |
| Execution Cycles | 27,868,519 | 25,906,422 | 28,797,895 | 25,828,113 |
| Compute Cycles | 11,337,782 | 11,290,799 | 11,337,671 | 11,301,515 |
| Load Operations | 1,832,392 | 1,821,846 | 1,838,008 | 1,832,174 |
| Store Operations | 744,111 | 585,998 | 766,181 | 579,291 |
| Idle Cycles | 16,530,737 | 14,615,623 | 17,460,224 | 14,526,598 |
| Cache Miss Rate | 5.86% | 5.54% | 6.15% | 5.47% |

**1.2. Benchmark 2 - blackscholes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Processor 0 | Processor 1 | Processor 2 | Processor 3 |
| Execution Cycles | 16,004,825 | 15,961,856 | 16,541,462 | 15,984,810 |
| Compute Cycles | 10,430,314 | 10,383,276 | 10,430,338 | 10,394,904 |
| Load Operations | 1,489,888 | 1,485,857 | 1,492,629 | 1,493,736 |
| Store Operations | 1,007,461 | 1,004,611 | 1,016,428 | 1,009,391 |
| Idle Cycles | 5,574,451 | 5,578,580 | 6,111,124 | 5,589,906 |
| Cache Miss Rate | 1.47% | 1.46% | 1.67% | 1.48% |

**1.3. Benchmark 3 - bodytrack**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Processor 0 | Processor 1 | Processor 2 | Processor 3 |
| Execution Cycles | 34,225,416 | 34,210,548 | 18,354,896 | 33,843,525 |
| Compute Cycles | 17,729,254 | 17,120,545 | 17,556,877 | 17,140,113 |
| Load Operations | 2,380,720 | 2,388,005 | 74,523 | 2,416,052 |
| Store Operations | 889,412 | 899,247 | 43,175 | 908,867 |
| Idle Cycles | 16,496,162 | 17,090,003 | 798,019 | 16,703,412 |
| Cache Miss Rate | 5.67% | 5.61% | 7.08% | 5.59% |

**1.4. Bus Statistics**

|  |  |  |  |
| --- | --- | --- | --- |
|  | FluidAnimate | Blackscholes | Bodytrack |
| Data Traffic (BYTES) | 29,606,720 | 6,001,792 | 20,694,624 |
| Invalidations | 2365 | 264 | 1877 |
| Access to Private Data | 3.85% | 0.46% | 0.93% |
| Access to Public Data | 96.15% | 99.54% | 99.07% |

**2. Dragon Protocol Statistics**

**2.1. Benchmark 1 - fluidanimate**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Processor 0 | Processor 1 | Processor 2 | Processor 3 |
| Execution Cycles | 27,827,825 | 25,860,273 | 28,791,348 | 25,778,025 |
| Compute Cycles | 11,337,782 | 11,290,799 | 11,337,671 | 11,301,515 |
| Load Operations | 1,832,392 | 1,821,846 | 1,838,008 | 1,832,174 |
| Store Operations | 744,111 | 585,998 | 766,181 | 579,291 |
| Idle Cycles | 16,490,043 | 14,569,474 | 17,453,677 | 14,476,510 |
| Cache Miss Rate | 5.86% | 5.52% | 6.15% | 5.46% |

**2.2. Benchmark 2 - blackscholes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Processor 0 | Processor 1 | Processor 2 | Processor 3 |
| Execution Cycles | 15,997,658 | 15,959,679 | 16,533,287 | 15,967,046 |
| Compute Cycles | 10,430,314 | 10,383,276 | 10,430,338 | 10,394,904 |
| Load Operations | 1,489,888 | 1,485,857 | 1,492,629 | 1,493,736 |
| Store Operations | 1,007,461 | 1,004,611 | 1,016,420 | 1,009,391 |
| Idle Cycles | 5,567,344 | 5,576,403 | 5,572,142 | 5,572,142 |
| Cache Miss Rate | 1.47% | 1.46% | 1.67% | 1.48% |

**2.3. Benchmark 3 - bodytrack**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Processor 0 | Processor 1 | Processor 2 | Processor 3 |
| Execution Cycles | 34,203,937 | 31,189295 | 18,353,499 | 33,822,244 |
| Compute Cycles | 17,729,254 | 17,120,545 | 17,556,877 | 17,140,113 |
| Load Operations | 2,380,720 | 2,388,005 | 74,523 | 2,416,052 |
| Store Operations | 889,412 | 899,247 | 43,175 | 908,867 |
| Idle Cycles | 16,474,683 | 17,068,750 | 796,622 | 16,682,131 |
| Cache Miss Rate | 5.65% | 5.61% | 7.06% | 5.57% |

**2.4. Bus Statistics**

|  |  |  |  |
| --- | --- | --- | --- |
|  | FluidAnimate | Blackscholes | Bodytrack |
| Data Traffic (BYTES) | 24,368,512 | 5,172,608 | 18,690,304 |
| Invalidations | 0 | 0 | 0 |
| Access to Private Data | 3.52% | 0.39% | 0.62% |
| Access to Public Data | 96.48% | 99.61% | 99.38 |

**3. Quantitative Analysis**

**3.1. Comparing Across Benchmarks**

In general, across the 3 benchmarks, there is greatest spatial and temporal locality of data in the blackscholes benchmark. This can be seen in Section 1.2 and 2.2 where the cache miss rates are around one-third of the cache miss rates of the other 2 benchmarks shown in Section 1.1, 1.3, 2.1 and 2.3. The low cache miss rate leads to the lowest number of invalidations and the least amount of data traffic as shown in Section 1.4 and Section 2.4. Both fluidanimate and bodytrack benchmarks seem to have the same degree of spatial and temporal locality in the data used as Section 1.1, 1.3, 2.1 and 2.3 show similar cache miss rate values.

In Section 1.3 and 2.3, Processor 2 seems to be an anomaly, executing a significantly lower number of operations then the other processors. Most of the instructions for the processor are of Label 2 which can be interpreted as computation operations, which explains the high ratio of its compute cycles to the total number of execution cycles as compared to the other processors. As such, load balancing could have been done on these 4 sets of instructions so that each processor finish at roughly the same time. This would reduce runtime of the benchmark as some of the workload can be shifted to Processor 2.

**3.2 Comparing Across MESI Invalidation and Dragon Update protocol**

Between these 2 protocols, the number of load and store operations between the same processor in the same benchmark will remain the same as it is determined by the trace files, which are the same across the 2 protocols. Generally, Dragon Update protocol performs better then the MESI Invalidation in a number of areas.

Firstly, under the Dragon Update protocol, processors have lower idle cycles while maintaining the same number of compute cycles. As such, under Dragon Update protocol, the processors were utilised more efficiently, which overall led to lower execution cycles of each processor, meaning that the runtimes of the benchmarks were also improved.

Secondly, under Dragon Update protocol, there were higher percentage of accesses to public data when we compare Section 1.4 and 2.4. In addition, there was lower data traffic, which means less reliance on the bus to carry data between caches or between the cache and the main memory.

**4. Literature Survey**

Improvement have been made to MESI over the years. A disadvantage of MESI is in the case of continuous read and write operations performed by various caches on a particular block, the data has to be flushed to the bus every time. Thus, the main memory will receive the updated value every flush and remain in a clean state. This challenge was overcome by MOESI protocol. The addition of an ‘Owned’ state in MOESI avoids the need to write a dirty cache line back to main memory when another processor tries to read it. Instead, the Owned state allows a processor to supply the modified data directly to the other processor. This is beneficial when the communication latency and bandwidth between two CPUs is significantly better than to main memory.

Another improvement from MESI was MESIF. The addition of the ‘Forward’ state in MESIF specifies one of the sharers of a clean cache line to respond to a read request for data using a direct cache-to-cache transfer instead of waiting for the data to come from the main memory. This optimization makes sense in architectures where the cache-to-cache latency is much smaller in comparison to the latency of accessing the main memory.

**5. Conclusion**

Cache coherence is essential in ensuring that the correct data is used, ensuring correctness of programs. In this assignment, I have explored the snooping mechanisms. There are 2 types of protocol under snooping mechanisms – wrote-through and write-back. Both MESI and Dragon Protocol are considered a write-back protocols.

Generally, wrote-through simplifies the design of the architecture. With write-through, the main memory always has an up-to-date copy of the line. Hence, when a read is done, the main memory can always reply with the requested data. If write-back is used, sometimes the up-to-date data is in a processor cache, and sometimes it is in main memory. If the data is in a processor cache, then that processor must stop main memory from replying to the read request, because the main memory might have a stale copy of the data. This is more complicated than write-through. The benefit of write-back protocols is that the latency between caches is much lower than the latency between the cache and the main memory. Hence, it takes lesser time to transfer to block from one cache to another, improving runtimes of programs.

There is no one-size-fits-all solution due to different characteristics of different workloads. Different workloads might perform better with different cache coherence techniques. Design of the whole processor also plays a part in determining the optimal cache coherence technique. For example, Intel uses MESIF protocol instead of MOESI because of its design. Intel's use of a large tag-inclusive L3 shared cache as a backstop for coherency traffic is their solution to the same problem that MOESI solves. As such, this removes the need to use MOESI protocol and partially explains why Intel use MESIF protocol instead. In conclusion, there are various factors that need to be considered in choosing the optimal cache coherence technique.