EECS 645: Coding Project Report

**Cache Coherence Simulations**

*Benjamin Wyss*

**KUID: 2898025**

*Date: 05/12/2019*

**Introduction**

This project serves to deepen my understand of caches and cache coherency by simulating the invalidation-based MOESI cache coherence protocol on a sequence of processor memory traces. To accurately simulate these processor traces, code that is detail-oriented and carefully designed must be implemented.

**Implementation and Design**

To tackle this challenge, I decided to write my implementation in Python since its compact syntax would allow me to focus more on design and less on syntax. My first major design choice was to abstract data into as many classes and objects as I could. I created separate classes to keep track of information about processor trace records, cache lines, and statistics regarding each processor. This design choice required a lot of setup on my end, but it made the implementation of the MOESI protocol run smoother.

The CacheRecord class I created is used to store information regarding each processor’s trace file. It keeps track of each record’s clock cycle, processor number, whether the processor read from or wrote to memory, and the memory address that the processor accessed. It then uses this information to calculate the offset, index, and tag that correspond to each memory address using the following calculations: Since each cache line in the direct-mapped cache is 32 bytes, then the offset must be the rightmost lg(32) = 5 bits of the memory address. Next, since the cache is 16 kilobytes in size, the total number of indexes must be 16 kilobytes / 32 bytes = 512 unique indexes – this corresponds to the lg(512) = 9 bits just to the left of the offset within the memory address. Finally, since each memory address is 32 bits long, this leaves the remaining leftmost 32 bits – 9 bits – 5 bits = 18 bits of the memory address for the cache tag. Additionally, the CacheRecord class is defined so that each processor trace record can be sorted chronologically by first comparing two records’ clock cycles and then by comparing their processor numbers.

The CacheLine class that I designed is used to keep track of information that concerns the MOESI states and memory tags that are currently stored in each processor’s cache lines. Each cache line is initialized with every processor holding no tags and beginning in the invalid state. This class simply serves as a data structure for my main program to interact with.

The ProcessorStats class that I created is another helper class which is utilized to record statistics that are reported at the end of the cache coherency simulations. Each ProcessorStats object records how many cache-to-cache transfers each cache makes, as well as the number of invalidations from each state and the total amount of dirty writebacks. This class also serves as a data structure for my main program to interact with.

The CacheCoherence class is the highest-level class which utilizes instances of CacheRecords, CacheLines, and ProcessorStats to simulate the MOESI cache coherence protocol and keep track of all necessary statistics. In order to maintain the accurate execution order of processor records, I first read in every trace file and stored each row inside of a list of CacheRecord objects, and then I sorted this list so that it would be in chronological order from beginning to end. This accomplishes the same behavior as incrementing a clock cycle from 1 to the end of the trace files, but it is more efficient since this implementation immediately skips past clock cycles where no memory was accessed. Once all CacheRecords are read in and sorted, the program functions by looping through each CacheRecord object, and then updating the corresponding CacheLine object and ProcessorStats object as needed. The implementation of this main loop is described as follows: First, the memory tag that is being placed into the cache line is compared with any currently existing tag to check if a conflict miss has occurred. If the new tag does not match the old tag, then a conflict miss occurs and is handled by invalidating the old tag and directly replacing it with the new tag, since the cache I am implementing is directly-mapped. If this conflict miss invalidates a cache line in the modified or owner state, then a dirty writeback must be recorded since those states are not consistent with main memory. After resolving any conflict misses, the new memory tag is placed into the cache line and then the processor’s state at that cache line is updated according to the MOESI protocol state diagram. After the processor’s state has been updated, it simulates generating a bus signal by calling the appropriate bus signal method (busRd, busRdX, busUpgr, or none). My busRd method simulates a read signal by searching for another processor to provide the cache-to-cache transfer and then updating the corresponding ProcessorStats object accordingly. The busRdX method simulates an exclusive read signal by looping through each other processor and invalidating its cache line if it shares the same memory tag. This updates the corresponding ProcessorStats object’s invalidations counter, as well as its dirty writebacks and cache-to-cache transfer counters whenever this signal would trigger a data flush. Lastly, the busUpgr method simulates a bus upgrade signal by looping through each other processor and simply invalidating each cache line that shares the same memory tag. Since this signal doesn’t trigger any data flushes, the corresponding ProcessorStats object is only updated in terms of its invalidations counter. After all CacheRecords have been simulated through my MOESI protocol loop, the program loops through each cache line at the end to write back any remaining dirty cache lines. Once this has completed, the results of the simulations are outputted to the console.

**Results and Discussion**

Upon executing the final program, the following statistics are outputted to the terminal:

P0 cache transfers: <p0-p1> = 51, <p0-p2> = 27, <p0-p3> = 27

P1 cache transfers: <p1-p0> = 15, <p1-p2> = 44, <p1-p3> = 44

P2 cache transfers: <p2-p0> = 7, <p2-p1> = 9, <p2-p3> = 43

P3 cache transfers: <p3-p0> = 20, <p3-p1> = 9, <p3-p2> = 4

P0 Invalidation from: m = 1, o = 8, e = 2, s = 0, i = 0

P1 Invalidation from: m = 4, o = 9, e = 2, s = 0, i = 0

P2 Invalidation from: m = 5, o = 4, e = 8, s = 0, i = 0

P3 Invalidation from: m = 5, o = 1, e = 11, s = 10, i = 0

Dirty Writebacks: P0 = 104, P1 = 115, P2 = 46, P3 = 60

Final States for P0: m = 9, o = 90, e = 11, s = 27, i = 375

Final States for P1: m = 17, o = 84, e = 28, s = 54, i = 329

Final States for P2: m = 2, o = 34, e = 30, s = 70, i = 376

Final States for P3: m = 40, o = 15, e = 6, s = 83, i = 368

All things considered, these statistics are believable, since a 64-kilobyte cache is not very likely to have tons of overlapping coherency and conflict issues during the short amount of program runtime that was provided in the trace files.

Looking back on my design and results, the greatest challenge in creating this project was the precise attention to detail required to maintain perfectly accurate cache simulations. For example, I originally encountered an issue where searching for a processor to provide a cache-to-cache transfer would simply search for and use the first processor that had the correct memory tag, but this would often be a processor in the shared state even though the transfer is supposed to come from a processor in the owner state. This issue required extensive manual testing and printing cache lines in order to discover what was happening, and it only impacted the final program by slightly altering the final processor statistics.

Another challenge that I encountered in this project was correctly handing different processor cache lines that store different memory tags at the same index. At first, this case seemed extremely confusing, since it would be possible for two different cache lines with the same index to have two processors in the modified state, the only difference being in the memory tag that they store. Ultimately, this issue was handled by sending a tag alongside all bus signal methods so that other processors’ MOESI states were only updated if they contained the same tag as the processor who generated the bus signal.

In the end, these challenges were overcome, and the results of my cache coherence simulations are able to speak of the attention to detail and precise execution that my program maintained.

**Conclusion**

Overall, this project has greatly aided my understanding of how a processor’s cache functions and handles difficult scenarios, such as those involving both conflict and coherency issues. Further, I have gained a deep and intuitive understanding of the MOESI cache coherence protocol, and the MOESI state diagram which used to appear complex and confusing now seems simple and efficient.