Computer Architecture Brandon Pramann, Daniel Hidalgo

EECS 645 KU id# 2966435

Final Project

Our approach to creating a simulation of cache coherence was to create a virtual world where for every object in real life there was an object in the code. We started by studying the interactions of the processor and the caches and how they used the bus to share or request data. Then we studied and recreated the state machine to make sure that we properly understood what we were trying to implement. After that, we did calculations to figure out the size of each of the components in our model. We choose to do the 32 KB cache, with 64 byte cache-line and 2 way LRU replacement policy for all processors. This meant that we would have 2 ^ 8 number of lines. Our index would therefore be 8 bits and our offset would be 6 bits. This would leave our tag with 18 bits.

From here we parsed through the trace files and broken up each line into three separate pieces. The first piece was the time of occurrence, and because the four files represent four processors running simultaneously we sorted the data chronologically. The second field was stored as an array of Booleans used to represent an IO operation, and finally we had an array of memory addresses which were converted to binary. Once converted to binary we further broke apart the address data created into an array of tags, an array of indexes, and an array of offsets.

A model computer was created to take that input data and process it based on the sorted chronological order. A processor ID was matched with the first available job and execute instructions based on the MOESI protocol. After the instruction had run, the processor would then pop that data of the front of every trace array it had. This would allow the computer to run the instructions in the order that the time stamps provided. In order to run an instruction based off the data, it would first check to see if the data was read or write. If the data was write, it would send a signal over the bus to invalidate the other caches data before it would write the data to it’s own cache and change its own state. If the data was read, the processor would first check to see if it had the data on it’s own cache before it would check the other processors to see if they had the data. If the processor had the data, nothing in our model would change. But if the processor did not have the data and other processors did, it would get the data from those processors and then it would increment the counter that was keeping track of how many times that the caches transferred data between itself. If the processor had no other choice other than to get the data from the memory, it would do so. After all these cases, it would change the state machine for the cache data appropriately. Then it would send a signal on the data bus if that was appropriate. The processors receiving the signal would then check if they needed to change states based off the signal they received. If they changed to invalid from any state other than invalid, the counter for the number of invalidations for that processor and state type was incremented. If the state for the cache line was changed to invalid from Modified or Owner, the counter for the number of dirty writebacks was incremented for that processor. In this way the computer would iterate through every command for each processor. After the computer had finished running all the processor commands, it would then display the results to the console.

To build and run the code, open the “EECS\_645\_Project.sln” file in visual studio on a windows computer. Run the program using “Ctrl + F5” and the results will be displayed to the console. Here is a visual copy for your enjoyment.: