Kyle Curry
Ethan Lefert

Final Project Report

For the implementation of the final project, we decided to use C++. Our project is composed of 11 files. Of these 11 files, 5 of them are files that we coded ourselves. 4 of the files are used to supply the data used for running the simulator. There is a read file that gives instructions on how to compile and run the simulator and finally a makefile that is used for compiling and running the lab.

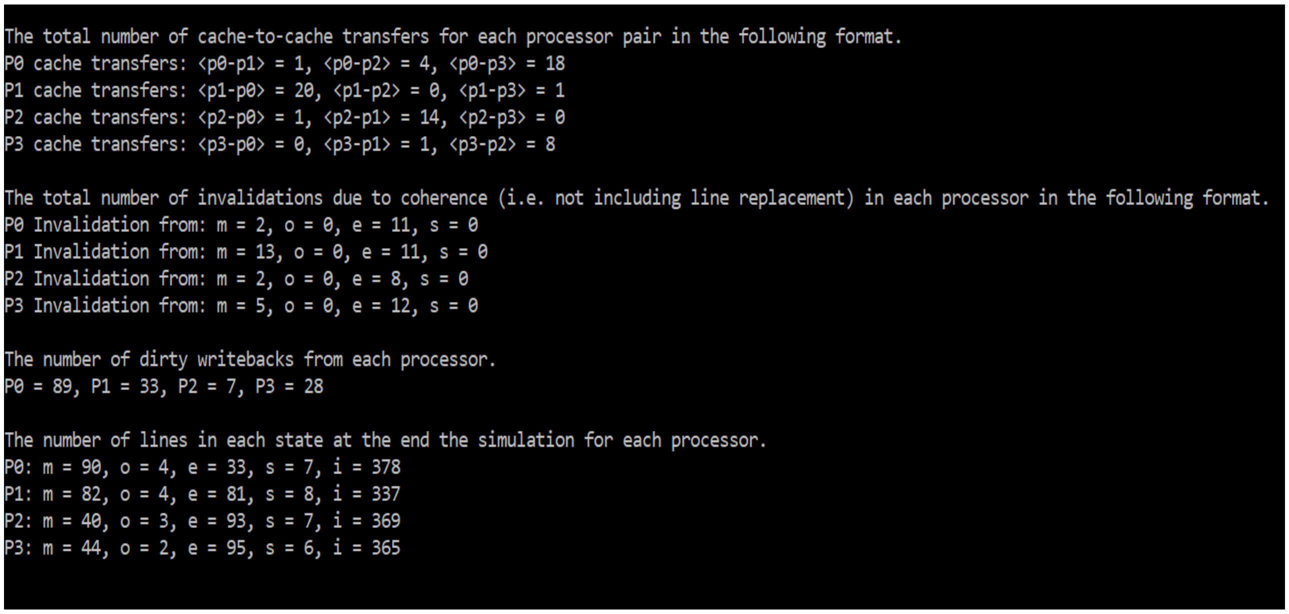
Of the 5 files, 2 of them are used for creating the processors. In the first processor file, we designed the processor object itself. Inside this file, we declared all of the variables that are used to keep track of the various statistics that are needed for the final output. We also declared all of the functions used for incrementing and returning these statistics. This file was also used to create the states array and the tag array. These arrays had a 1 to 1 association and were both made to hold 512 items. The states array kept track of the current state at that same index in the tag array. Finally, this file was used for declaring the functions that were used to change from one state to another.

The second processor file was used to initialize the variables to their starting values. We also used this file to increment some of the variables while the simulator is running and to get the final count of other variables when the simulation has ended. Lastly, this file initialized the tag array to empty values and the states array had every index set to the invalid state.

The second set of files that we wrote for this lab were used to read in the files, parse the values, and finally sort them. The first file was used to create the parser class. Inside this class we declared 4 arrays, each used to track a different value. One array was the processor array used to keep track of the current processor number. The next array used was the cycles array to keep track of the cycle count. Another array was used to keep track of the read/write bit. The last array was used to keep track of the hex value for each line of the input file. Each array was the same maximum size, 3456, which was the total number of lines of the 4 data files. In addition to these arrays, we created a size variable used to keep track of the current size of each array. Lastly we declared functions used for sorting the data and retrieving information from the arrays.

The second parser file is where the actual parsing and sorting happen. First, the initial size is set to 0. Then there are 4 parsing functions, one function per input file. These functions read through each file line by line and assign every item to a specific array. Once all 4 files were done being read in and the array has been filled, the sort then happens. We used a selection sort that would go through the cycles array and sort it based off of the cycle number, the lower cycles went first and the higher numbered cycles went last. If there were 2 cycles of the same number, priority was given to the processor with the lower number. When a swap occurred, we used a swap function on all 4 arrays so that when an item swapped in the cycles array, it was swapped in the other 3 arrays.

The last file that we wrote was the main file where the bulk of the program is run. First in the main file we have a function for converting a hex value to a binary value. This binary value is used later inside our function that gets the index value and is also used to get the tag value. Most of the main file is executed within a for loop that runs 3456 times, which is again the total number of lines from the input file. Inside the for loop, we have many if/else statements for checking various conditions on each run through the loop. First, we check the current read/write bit. Depending on if we currently have a read or a write, we go to a different outcome. If we are currently reading, then we go inside the read branch and then check to see what the current processor is. Once we have determined the current process, we then check to see if the current tag is in the tag array currently and if we are currently at an invalid state because then we do nothing. If however the tag isn't in the array, we then check to see what our current state is so that we can update state and respond accordingly. At the very end if we aren't able to find the current tag, we then add it to the tag array. If we currently have a write bit instead of a read bit, we first check what the current processor is. Once we have determined the current processor, we then find out what our current state is so that we can react accordingly. Finally, at the end of main, we have several output statements to print the final statistics to the terminal screen.



```
The total number of cache-to-cache transfers for each processor pair in the following format.
P0 cache transfers: <p0-p1> = 1, <p0-p2> = 4, <p0-p3> = 18
P1 cache transfers: <p1-p0> = 20, <p1-p2> = 0, <p1-p3> = 1
P2 cache transfers: <p2-p0> = 1, <p2-p1> = 14, <p2-p3> = 0
P3 cache transfers: <p3-p0> = 0, <p3-p1> = 1, <p3-p2> = 8

The total number of invalidations due to coherence (i.e. not including line replacement) in each processor in the following format.
P0 Invalidation from: m = 2, o = 0, e = 11, s = 0
P1 Invalidation from: m = 13, o = 0, e = 11, s = 0
P2 Invalidation from: m = 2, o = 0, e = 8, s = 0
P3 Invalidation from: m = 5, o = 0, e = 12, s = 0

The number of dirty writebacks from each processor.
P0 = 89, P1 = 33, P2 = 7, P3 = 28

The number of lines in each state at the end the simulation for each processor.
P0: m = 90, o = 4, e = 33, s = 7, i = 378
P1: m = 82, o = 4, e = 81, s = 8, i = 337
P2: m = 40, o = 3, e = 93, s = 7, i = 369
P3: m = 44, o = 2, e = 95, s = 6, i = 365
```