Write-Up

Anjin

Dhruv- Worked on the LRU hit and miss functions(not implemented versions), performance assessment and write up

Hao- Worked on branch prediction and implementing the push\_pipeline\_stage function

Jay- Worked on cache data structure and associated methods, some editing

In our project we implemented several methods which allowed our pipeline to run accurately. First we ask the user to enter in a cache index, block size and associativity. We then make sure that we don’t not exceed the cache size of 10240. Next we dynamically create the cache and initialize the pipeline. We then go through and parse the information in our trace file.

Every time we parse one line, we check to see if the address is already in our cache, using iplc\_sim\_trap\_address. If it is in our cache, then we call the iplc\_sim\_LRU\_update\_on\_hit method which will simply add to our counter for the amount of cache hits and update the value in our cache. However, if the address is not in our cache then we call the iplc\_sim\_LRU\_replace\_on\_miss which goes through and looks to find either an empty spot in the cache or looks to find the lowest value in the cache and replace the old data with the new address’s data. We then also update the amount of cache misses. If we do have a miss, in our iplc\_sim\_parse\_instruction method we push the instruction through the pipeline. The purpose of doing this is that we do not double count cycles but instead have them overlap.

In the pipeline simulator, iplc\_sim\_push\_pipeline\_stage, we go through seven stages. The first stage is the writeback stage in we write our results into the register file. The second stage we check to see if the branch prediction was right. The third stage is to check for any LW delays. If there are any then we add 9 cycles to our cycle count. We are adding one additional cycle later one so that is why we do not add the full ten cycle delay. In the fourth stage we look for a SW memory access and data miss and add 9 cycles if it does happen. In the fifth date we just implement the total cycles by one. In the sixth stage we push through onto the next stage of the pipeline. In the seventh stage we reset the fetch stage to NOP.

We then go back into the iplc\_sim\_parse\_instruction and parse the instruction. We check to see if then instruction if “add”, “sll” or “ori” as if it is then we assign values to our dest\_reg, src\_reg and src\_reg. We then call the iplc\_sim\_process\_pipeline\_rtype method which basically assigns the dest\_reg, src\_reg and src\_reg to the Fetch in the pipeline. Next, we check to see if the instruction is “lui”, if it is we then call the iplc\_sim\_process\_pipeline\_rtype method again but this time we would assign the same value to dest\_reg as the last conditional but assign the values -1 to both src \_reg and src\_reg2. We then check to see if the instruction is “lw” or “sw”, if it is “lw” we then set the dest\_reg value and call the iplc\_sim\_process\_pipeline\_lw method which essentially sets the values of the dest\_reg, base\_reg and data\_address for the lw in the pipeline. If the instruction was “sw” however, then we assign the proper value to the src\_reg and call, iplc\_sim\_process\_pipeline\_sw. If the instruction is “beq” then we call iplc\_sim\_process\_pipeline\_branch. If the instruction was “jal”, “jr” or “j” then we call iplc\_sim\_process\_pipeline\_jump. If the instruction is “syscall” we call iplc\_sim\_process\_pipeline\_syscall. If the instruction is “nop”, then we call iplc\_sim\_process\_pipeline\_nop. All of these methods just assign values to the targeted pipeline. Lastly, we call iplc\_sim\_finalize which will ensure all instructions are done procceded and print important statistics.

Performance Assessment

Based on the results of our 18 different tests. We have concluded that the best highest performing configuration is the one with a cache index, Block Size and Associativity of 4 with the branches not taken. This can be proved by 3 key statistics.

First, we will talk about CPI. CPI is the average number of CPU clock cycles that occur per an instruction are being executed. It can be calculated by taking the total amount of cycles and dividing it by the amount of instructions.

As we can see the 9th bar on the bar graph is below the average. The average CPI for all 18 tests was 1.248122389 while the CPI of this specific configuration is 1.057491. The standard deviation for this data was 0.135008803. Thus, we know that the CPI of 9th bar is well below one standard deviation meaning that’s its CPI is in about the lowest 13 percentile. A low CPI is good because you want the least amount of cycles to pass per instruction, meaning that it takes less time to complete and instruction.

Second, we will discuss the miss rate. The miss rate is fraction of memory references not found in the cache. It can be calculated by dividing the cache misses by the cache accesses.

We can see that the 9th configuration had an extremely low miss rate to many of the other configurations. The average miss rate was 0.015827444 while the miss rate for the 9th configuration was 0.002175. The standard deviation of the data is 0.012806946, thus the 9th configuration is more than a standard deviation away from the average meaning that again, it is in the lowest 13 percentile. Having a low miss rate is good because that means that you spend less time copying things from the main memory to the cache.

Lastly, we would like to bring up the statistic of the total correct branch predictions. This basically means how many times did our program guess the right way to branch off to.

This data clearly shows that the branch not taken method usually yielded better result than the branch taken method by a lot.

These three statistic give us plentiful evidence that the 9th configuration performs because it has one of the lowest CPI’s and miss rates and has a higher correct branch prediction rate than the branch taken method configurations.

Here is the full data chart:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Configuration Number** | **Cache Index** | **Block Size** | **Levels of Associativity** | **Total Cycles** | **Total Instruction** | **Total Branch Instructions** | **Total Correct Branch Predictions** | **CPI** | **Number of Cache Accesses** | **Number of Cache Misses** | **Number of Cache Hits** | **Cache Miss Rate** |
| 1 | 7 | 1 | 1 | 48547 | 34753 | 7044 | 5730 | 1.396915 | 35863 | 1390 | 34473 | 0.038759 |
| 2 | 6 | 1 | 2 | 40556 | 34753 | 7044 | 5739 | 1.166978 | 35863 | 502 | 35361 | 0.013998 |
| 3 | 5 | 1 | 4 | 39309 | 34753 | 7044 | 5744 | 1.131097 | 35863 | 363 | 35500 | 0.01022 |
| 4 | 6 | 2 | 1 | 47757 | 34753 | 7044 | 5276 | 1.374184 | 35863 | 1301 | 34562 | 0.036277 |
| 5 | 5 | 2 | 2 | 39261 | 34753 | 7044 | 5736 | 1.129715 | 35863 | 357 | 35506 | 0.009955 |
| 6 | 4 | 2 | 4 | 37848 | 34753 | 7044 | 5746 | 1.089057 | 35863 | 200 | 35663 | 0.005577 |
| 7 | 6 | 4 | 1 | 42979 | 34753 | 7044 | 5729 | 1.236699 | 35863 | 770 | 35093 | 0.021471 |
| 8 | 5 | 4 | 2 | 37345 | 34753 | 7044 | 5749 | 1.074583 | 35863 | 144 | 35719 | 0.004015 |
| 9 | 4 | 4 | 4 | 36751 | 34753 | 7044 | 5749 | 1.057491 | 35863 | 78 | 35785 | 0.002175 |
| 10 | 7 | 1 | 1 | 52998 | 34753 | 7044 | 1279 | 1.524991 | 35863 | 1390 | 34473 | 0.038759 |
| 11 | 6 | 1 | 2 | 45015 | 34753 | 7044 | 1280 | 1.295284 | 35863 | 502 | 35361 | 0.013998 |
| 12 | 5 | 1 | 4 | 43769 | 34753 | 7044 | 1284 | 1.259431 | 35863 | 363 | 35500 | 0.01022 |
| 13 | 6 | 2 | 1 | 52193 | 34753 | 7044 | 1290 | 1.501827 | 35863 | 1301 | 34562 | 0.036277 |
| 14 | 5 | 2 | 2 | 43707 | 34753 | 7044 | 1290 | 1.257647 | 35863 | 357 | 35506 | 0.009955 |
| 15 | 4 | 2 | 4 | 42304 | 34753 | 7044 | 1290 | 1.217276 | 35863 | 200 | 35663 | 0.005577 |
| 16 | 6 | 4 | 1 | 47417 | 34753 | 7044 | 1291 | 1.3644 | 35863 | 770 | 35093 | 0.021471 |
| 17 | 5 | 4 | 2 | 41803 | 34753 | 7044 | 1291 | 1.20286 | 35863 | 144 | 35719 | 0.004015 |
| 18 | 4 | 4 | 4 | 41209 | 34753 | 7044 | 1291 | 1.185768 | 35863 | 78 | 35785 | 0.002175 |