Assignment 4: Cache Modeling

Roger Harriss Brammall

27645143

November 24th, 2017

# Results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **RESULTS** | fppp | go | gcc | vpr |
| a\_i\_miss\_rate | 0.058665 | 0.025605 | 0.037723 | 0.005611 |
| a\_ii\_miss\_rate | 0.110599 | 0.029877 | 0.034102 | 0.000253 |
| b\_ld\_miss\_rate | 0.001596 | 0.005623 | 0.009752 | 0.077863 |
| b\_st\_miss\_rate | 0.000379 | 0.002991 | 0.003617 | 0.006072 |
| b\_wb\_ratio | 0.000127 | 0.002625 | 0.00302 | 0.009397 |
| c\_direct\_ld\_miss\_rate | 0.053344 | 0.035512 | 0.021471 | 0.03216 |
| c\_direct\_st\_miss\_rate | 0.071147 | 0.01009 | 0.010104 | 0.008461 |
| c\_2way\_ld\_miss\_rate | 0.008231 | 0.010857 | 0.011352 | 0.021218 |
| c\_2way\_st\_miss\_rate | 0.004878 | 0.004262 | 0.004259 | 0.006112 |
| c\_4way\_ld\_miss\_rate | 0.001718 | 0.007583 | 0.009797 | 0.01854 |
| c\_4way\_st\_miss\_rate | 0.000495 | 0.003194 | 0.003687 | 0.005898 |
| d\_miss\_rate | 0.055487 | 0.016241 | 0.021585 | 0.000142 |
| d\_prefetch\_success\_rate | 0.996474 | 0.919206 | 0.766721 | 0.934152 |

Figure 1: results from fppp benchmark

It can be seen that the cache’s with the most misses are the instruction fetch caches in fpppp. More so a 4-way associative cache (a\_ii) has a higher miss rate than a direct map cache. The store and load instructions seen in part b and c have a much lower miss rate because instructions probably use the same load and store addresses repetitively. It can be seen that a larger associativity assists in hit rate on instruction load and stores, which means that most likely the loads and stores happen in alternation around similar addresses. A comparison between caches with and without victim caches for part c showed that the more associative a cache is the less impact a victim cache has on the result, this analysis was only taken using the fpppp benchmark and is assumed to be similar throughout the rest of the bench marks. The prefetch cache added in part d helped significantly with the miss rate of the 4-way associative cache used in part a\_ii. This means that many of the instructions occur in a sequential order in fpppp.

Figure 2: results from gcc benchmark

The gcc benchmark test in general had a lower miss rate than the vpr and fpppp benchmarks with a maximum that is less than 3.8% miss rate in the cache. This means most likely that this code is executed relatively sequentially and with small loops. Smaller loops executing the same instructions tend to have cache hits due to using the same data over and over. The prefetch cache implemented in part d only had a success rate of 76% which can be seen to have made less of a difference between part a\_ii and part d than the other four benchmarks. In this case the direct map cache is less accurate for instruction holding than the 4-way associative cache which may be an indication that there are many jumps in the execution, which also explains the lower usage of the prefetched instructions in part d.

Figure 3: results from go benchmark

Go is similar to gcc in that it has a lower cache miss rate hinting that it most likely uses small loops and jumps that use the same instructions repeatedly compared to fppp and vpr. However something to note is that direct map cache is better than the 4-way associative cache for the instruction cache. This can be due to jumps and loops that are commonly outside the bounds of the smaller index in the 4-way cache but are within the index size of the direct map cache. The higher prefetch use lowered the miss rate of the 4-way cache significantly meaning the most likely the instructions being fetched were just outside of the index of the 4-way cache, while also the instructions happened relatively sequentially in terms of address. Another major difference is the large amount of load misses in the direct cache for load and store instructions. This can be due to instructions commonly loading to the same tag in alternation.

Figure 4: results from vpr benchmark

The largest miss rate is the direct cache in part c. This might be due to loading from address multiples of the index apart from each other. It can be seen that instruction miss rate is very small in vpr so most likely the instructions run in small loops and change the addresses of the store and load instructions in each iteration of the loop.

# Part A

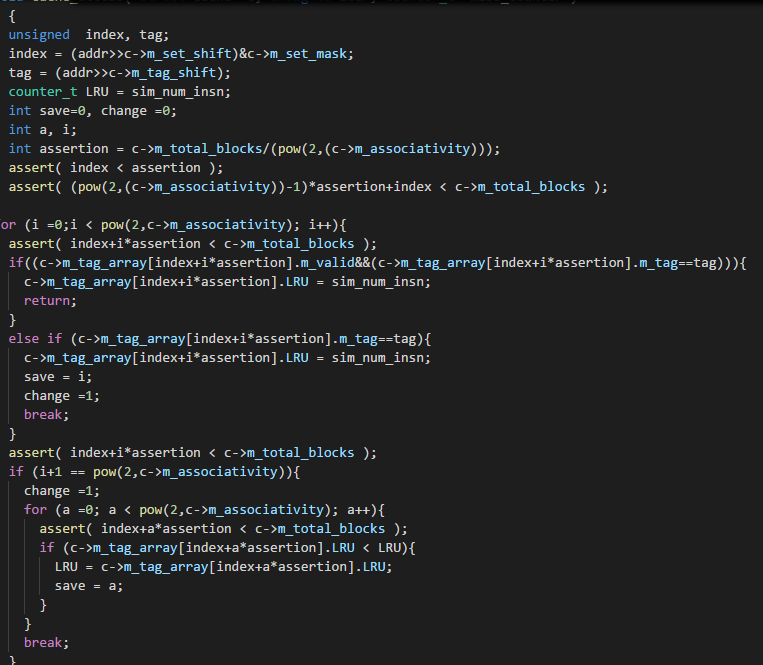


Figure 5: code used for part a i and ii

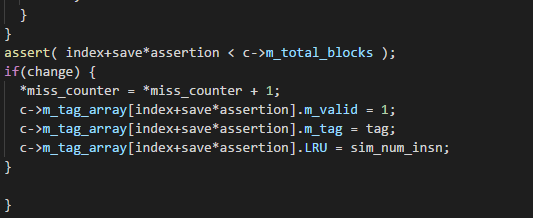


Figure 6: code used for part a i and ii

Using nested four loops that tags were queried at each index to find a tag that matches the current instruction fetch. After iterating the first for loop to search for a matching tag, if none were found, then the next for loops looks for the LRU cache value. It flags for that value to be changed and enters the if(change) portion and adds to the miss counter and changes the data in the cache to the new data.

# Part B

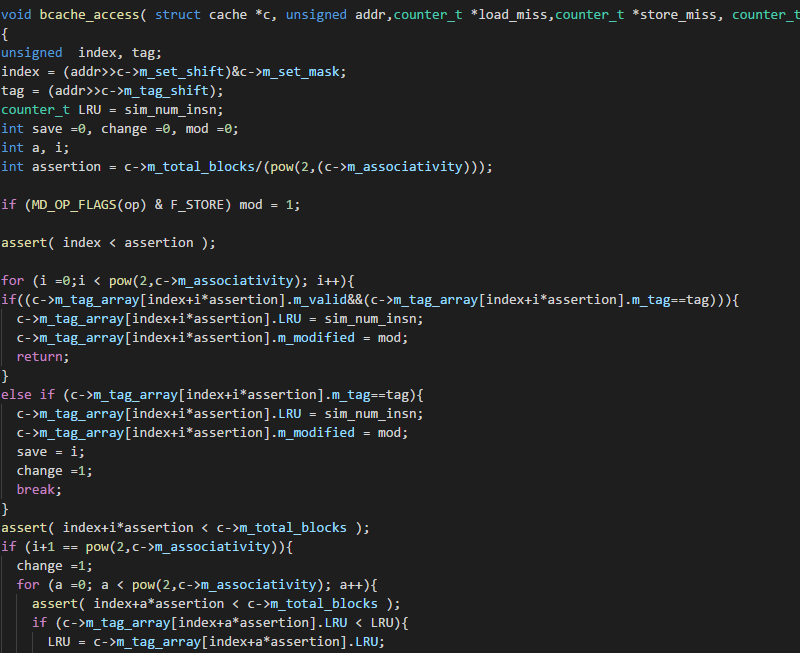


Figure 7: code used for part b

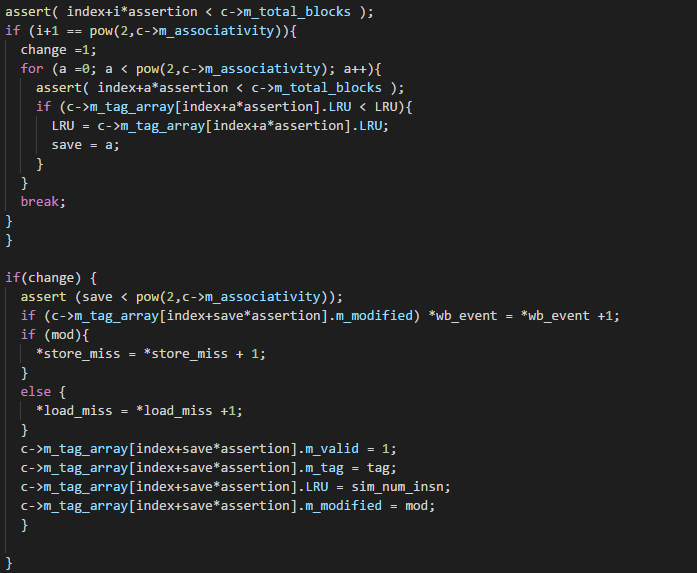


Figure 8: code used for part b

The major difference for the code in part b is that it takes the addr calculated from the #define READ\_BYTE function that is the target of the load or store instruction. Using an if statement to tell if the memory function is a load or store, mod is set to one when it is a store. If the address is found in the cache the block gets the modified flag. When there is a miss in the cache it checks to see if it is a store or load miss and then increments a writeback event when the data has the modify flag.

# Part C

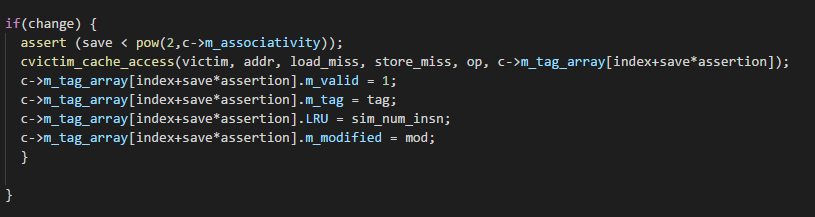


Figure 9: eviction portion of the main cache for part c

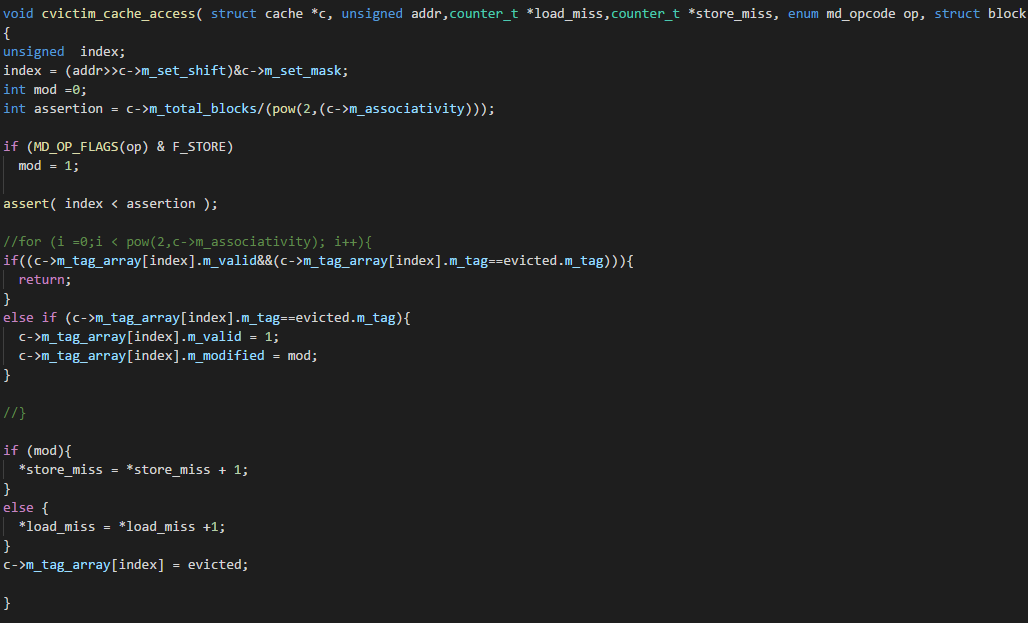


Figure 10: victim cache

The main cache runs the same as part b however when a miss occurs it runs a victim cache access and passes the evicted block to the victim cache. The victim cache checks to see if it contains the requested data (the data at addr) and if it has the data returns to the main cache without incrementing the miss counters but if it is a victim cache miss it will increment the miss counters accordingly.

# Part D

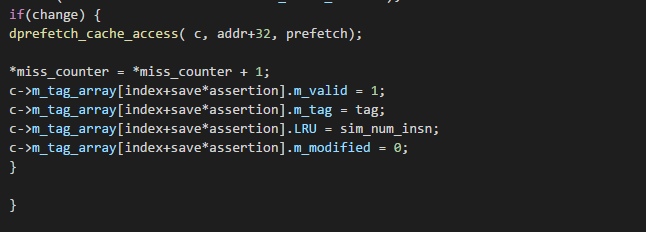


Figure 11: miss portion of the cache for part d

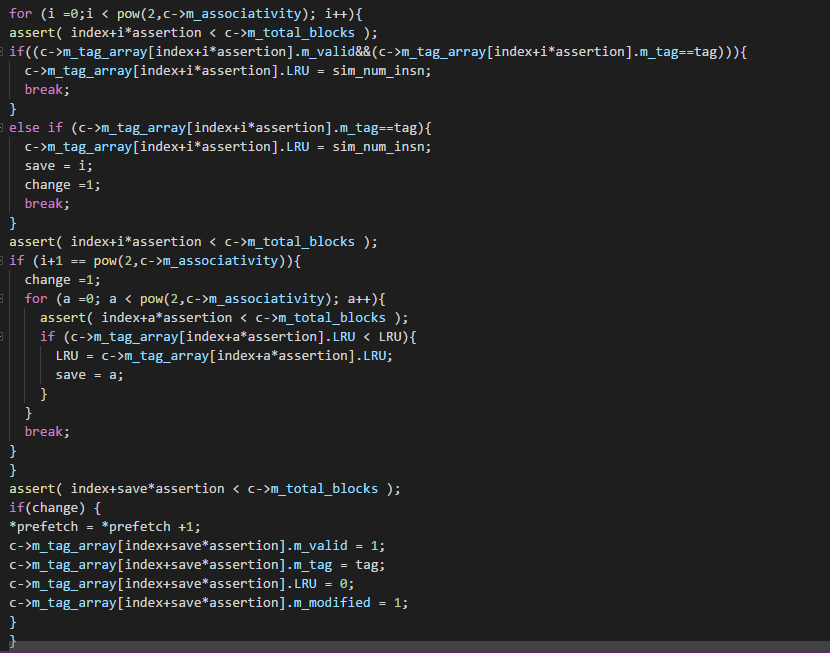


Figure 12: prefetch part of cache part d

When the main cache misses, seen in figure 12, it calls a secondary function that runs through the cache again. It checks for a full associativity and then fetches the next instruction (32 bytes away). It then sets the LRU to the least possible value so that if the prefetch is incorrect that data is removed immediately from the cache