CS 3853 Computer Architecture

Project Report

Monday, Dec 2, 2019 7:30 p.m.

Team\_14

By signing this report I affirm that I know and agree with the contents.

Signatures:

Name #1: \_\_\_\_\_\_\_\_\_\_Chris Crabtree\_\_\_\_\_\_\_\_\_\_\_\_\_

Name #2: \_\_\_\_\_\_\_\_Zachary Mauldin\_\_\_\_\_\_\_\_\_\_\_\_\_

**Objectives:**

1. **Comprehend how a cache implementation works**

We learned how the cache is created, maintained and all the calculations for determining cache efficiency. We also learned about the argparse and logging modules in the standard python library which assisted us in parsing the command-line arguments and debugging.

1. **Determine the performance benefits based on cache configuration**

Based on the cache configuration, a cache is a lot faster than disk, time can be used more efficiently and effectively. Increasing the cache size always seems to help. Likewise, increasing the associativity always seemed to help, although as the cache size grew the associativity seemed to matter less. For the block size, increasing it lowered the miss rate, but increased the cpi. If you care about the cpi most, then a low block size seemed to yield the best results. The replacement policy seemed to matter the least. With a small cache, high associativity, and small block size (the situation you would expect the replacement policy to matter the most) LRU only lowered the miss rate by .5% and the CPI by .1 over RR.

1. **Learn to work in a small group**

We spilt up the work throughout the milestones, and have put together this project.

We each kept up doing our parts. Version control implemented.

**Algorithm**

For our cache implementation we used dictionaries. The row number was the key to cache dictionary. Each row, in turn, contained it’s own dictionary with the keys being the tags for the blocks in it. The row dictionaries were forced to maintain a number of blocks equal to the associativity for the cache. The simulator algorithm reads a trace file, and if it is a destination address, send the address to the cache to determine where in the cache the data will belong. If it is a hit, the appropriate cycles are recorded and returned. If it is a miss, the replacement algorithm determines which tag of associativity >1 will be replaced, and then returns the cycles it took for the miss. With the total hits, misses, and cycles accrued during execution, calculations are done to calculate CPI, hit rate, miss rate, hits, misses, cache accesses.

**Analysis**

The first graph on the last page is from the file: “Trace2A.trc”. It shows how the different replacement algorithms affect the CPI. LRU and RR have a better affect on the CPI than RND. The next 4 graphs are from using our simulator on the file “Corruption1.trc”. These show varying block sizes and cache sizes along with different associativity. Based on our graphs, the change from associativity 1 to 2 makes a huge difference in the CPI. At associativity’s 4, 8, and 16, the change in CPI is significantly less, but improves. Also having a big cache size with a small block size doesn’t affect CPI for associativity greater than 2. Vise-versa having a small cache size with a big block size halts CPI from lowering after associativity of 8.

**Technical Issues:**

(Describe your most prominent technical issues and how you solved them. What was most difficult to figure out? You should have at least one and no more than three.)

The main issue was debugging. All of our high level conceptual designs ended up being correct, but with many moving pieces it was difficult to make sure everything was working properly in every case. This was especially true of the logical bugs, as they were not readily apparent until we tried to align our results more closely with Professor Ortiz’s program. Since we could not inspect or experiment with his program’s source code, we had to iteratively sift through decreasing subsets of the trace files provided until we singled out examples that produces differing results in our two programs. Once a bug was found, fixing it became quite simple but a lot of effort went into finding those examples.

**Group Member Contributions:**

Chris - Wrote most of the code for the project, debugging

Zac – Did some testing, and the report.

**Group Issues and Resolutions:**

None.

**Conclusion:**

This project was interesting, it definitely helped teach more about how the cache works and gives us a hands-on approach to the process of a cache.

**PROJECT FEEDBACK:**

While I (Chris) enjoyed the challenge of helping implement a cache in an accurate manner, I thought it was still too straightforward to find optimal cache configurations (i.e. it was always best to have high associativity and a large cache). I liked the addition of the miss penalty to constrain the block size, but according to the Wikipedia page on the subject (<https://en.wikipedia.org/wiki/Cache_placement_policies#Disadvantage_2>) there are disadvantages to having a high associativity as well. Since you mentioned several times in class that the ‘cost’ of large caches and high associativity was the overall physical size of the cache, I think it would be beneficial to implement some sort penalty associated with the cache’s physical size, as determined by the cache’s total implementation cost (in bytes, since that can be much higher just the cache size), the associativity (for the hardware needed check all sets in unison), and the replacement policy(which according to what I’ve read can affect physical size since extra hardware is needed to implement the LRU replacement policy). An example of this would be imposing an arbitrary bound on cache and main memory size so that as the cache gets bigger main memory has to get smaller (which would increase the likelihood of missing main memory and having to read from disk, thus greatly increasing the CPI). I’m probably one of the only students that would enjoy that addition, but I think it would give the students a deeper understanding of the constraints that need to be considered when designing a cache.