Caleb Hubbell

ECE 472

Final Project

Due 12/ 7

Implementing a Cache Simulator

**Part 1:**

**PH1:** *Summarize your work in a well-written report. The report should be formatted as a single-spaced document with 12pt text. Use images, charts, diagrams or other visual techniques to help convey your information to the reader.*

This cache simulator is a result of a final project for ECE 472. As an undergrad I was challenged to implement numerous cache features, including: N-set associativity, random replacement and a true Least Recently Used (LRU) scheme, and write-through. The cache accepts two input parameters: a configuration file, and a trace file. As the cache runs on the input, it will track the number of hits, misses, and evictions, as well as the total elapsed cycles simulated. This project was formed to solidify what we have learned in the class, and to give us an opportunity to practice our C++ coding.

I chose to center my cache implementation around vectors. Vectors in C are defined as “sequence containers representing arrays that can change in size” (C++.com) and are the perfect data structure to store a cache, since they allow direct element access, and dynamic resizing based on the configuration file parameters. My cache utilizes four separate vectors, three of which are two dimensional. I chose to keep my cache components separate, relating them to the index and “way” (or offset), as this led to a better understanding of referencing the cache, for both myself and the reader.

My cache structure is as follows:



Fig.1 Cache Structure

As mentioned above, this cache structure was decided upon because one can easily understand how each dataset is found based on the references besides it. For example: The cache needs an index, a way, and a tag, then the data. But, for the sake of this cache, we do not care about the data, and so it is omitted. Another interesting data structure choice is that of my LRU; I chose to utilize a list inside of a vector. The vector, like every other structure, is used to reference a certain index, then the list keeps track of the Least Recently Used way. The list simply always pushes the way to the front, and then, based on what has happened, will remove the appropriate way. The other benefit of the list is that it easy to tell when the cache fills up, as the list will return its size.

Most of the challenges I encountered were during the planning phase of the cache. I was familiar with how the cache worked, and what was necessary to create a cache, but that is half the battle. I spent a long time choosing the data structures, and then deciding the most intuitive way to represent the data. Once the planning and structuring phase was complete, filling the cache was a simple task.

While thought went in to how my cache was structured, as explained above, the numerous 2D vectors sometimes led to long reference strings. If I were to implement this cache differently, I think I would create a different cache structure. I would make own classes, as a way to simplify references. First, my cache would be a 1D vector which was referenced via index. Stored in the vector would be a custom class containing a full flag, and a list. This list would be of another custom class, holding a valid flag and a tag. With this custom setup one could easily hit the index, check if the way is full, search the list for the way, and receive back the tag and valid flag.

The keeping of clock cycles was another tricky part of the project. Two different amounts of time need to be stored: a cache access, and a memory access. The latter takes significantly longer. Once this was realized, a simple flow chart best demonstrates how to track the clock cycles:



Fig. 2 Clock Cycle Chart

As seen, my code does not hit all the cases, as I did not implement the extra credit write back.

**PH2:** *Explain how you implemented your cache simulator. You should provide enough information that a knowledgeable programmer would be able to draw a reasonably accurate block diagram of your program.*

* *What data structures did you use to implement your design?*
* *What were the primary challenges that you encountered while working on the project?*
* *Is there anything you would implement differently if you were to re-implement this project?*
* *How do you track the number of clock cycles needed to execute memory access instructions?*

**Part 2:** *There is a general rule of thumb that a direct-mapped cache of size N has about the same miss rate as a 2-way set associative cache of size N/2.*

*Your task is to use your cache simulator to conclude whether this rule of thumb is actually worth using. You may test your simulator using instructor-provided trace files (see the sample trace files section) or you may generate your own trace files from Linux executables (“wget oregonstate.edu”, “ls”, “hostid”, “cat /etc/motd”, etc). Simulate at least three trace files and compare the miss rates for a direct-mapped cache versus a 2-way set associative cache of size N/2. For these cache simulations, choose a block size and number of indices so that the direct-mapped cache contains 32KiB of data. The 2-way set associative cache (for comparison) should then contain 16KiB of data. You are welcome to experiment with different block sizes/number of indices to see how your simulation results are affected. You could also simulate additional cache sizes to provide more comparison data. After you have obtained sufficient data to support your position, put your simulation results into a graphical plot and explain whether you agree with the aforementioned rule of thumb. Include this information in your written report.*