University of Colorado
Colorado Springs

CS 4200, Spring 2022: Computer Architecture

Instructor: Mong Sim

Homework Four

Andrew Schmidt

2022/03/01

Project Overview

In my simulation of LRU and PLRU cache replacement we are given 96 different address given as hex numbers in an order that would simulate a CPU calling different address for a program. Depending on the address, we need to figure out if its loaded into our cache yet and mark it as a hit if its there, or a miss and use LRU or PLRU to replace the correct location in the cache with the data. Since there's no data being simulated here, we simply list the data for different address as data1…data4 since we are looking at 4 words at a time during address calls, as noted by bits 0 and 1 which will be assumed to be 0 for all address in the simulation. Since this is a 1:4 the data is read 4 blocks at a time, where all 4 data points have the same tag, and are differentiated by bits 2 and 3 allowing for each data block to be called upon independently, however we only care about hit/miss in the tag comparison so this information is thrown out. In a real CPU, this would be used to know which instruction/data to return. Next we set up 4 tags, which goes to 16 data points, for each structure that will be governed by its on cache replacement structure either LRU or PLRU, and then lastly we multiply this by 4 to get our 4x4 array of tags, with each row being governed by its own cache replacement structure. When the address is called, it will check all the tags in a column, which is determined by bits 4 and 5, and compares tags, along with checking the valid bit. If the data is valid and the tags match it calls a hit, which would return the data/instruction that the CPU asked for, then update the LRU/PLRU accordingly.

If it's a PLRU, the bits on the side of the tree, 0 and 1 for way 0 and 1, and bits 1 and 2 for ways 2 and 3, would be changed so that their side pointed away from the most recently used bit while leaving the 3rd bit alone in the state it was originally in. This means if the tree was pointing at way 0 the top bit, bit1, would point to the right = 1 and bit 0, far left of the tree, would point to way 1 so that when it comes around it sees way 1 as least recently used over way 0. This works alright, however doesn't consider that if way 0 is the most recently used and Way 1 is the least recently used, it will change out either way 2 or 3 instead of way 1 because way 0 was listed as MRU.

In LRU it simply updates the array for the cache line its in so that the current way is stored in temp, everything to the left of it is moved over and then the MRU is placed in array index 0. If it's a miss, it does the same process and replaces the way stored in index 3 since that holds the spot for the LRU way and all others get shifted over.

For each instruction it marks the data as hit or miss, fills in any data into cache that is needed, and then prints the data into a table labeling the sequence, Way, Valid bit, c.tag, c.data.index, c.tag.index, history stack, status, and the dummy c.data. **note: Since we update the fields first before* *printing, valid will always print as 1, and history stack is updated before printing*.* Both LRU and PLRU return the value of (hits/96 * 100) so the efficiencies of both can be compared to see the effectiveness of each. Below I have included an example of how the address is parsed and compared to validate a hit/miss, along with how it decides where to place the information in the event of a miss.

Result Analysis

Upon running the program, we see that LRU and PLRU are both very similar at 59.38% and 63.54% respectively. This is so close due to the program simulation being so short and being only 1 set of test instructions where real life examples are much longer and have a larger cache structure. Since these numbers should be taken with a grain of salt, we need to look at the other pros and cons of each replacement algorithm.

Implementing a LRU method is best for efficient cache since data needs to be swapped less resulting in lower power consumption, less heat produced, and faster run times as the CPU will stall less as it waits for data to be loaded in. While this is the best method to use, it takes much more overhead on storage as a 4-way set associative needs 8 bits and going into 8 ways would need 8*3 = 24bits. This overhead makes it impractical for larger n way's, along with the added complexity in hardware to handle these conditions.

Implementing PLRU is worse off in most situations as it doesn't track the history of data use as well, often leading to the LRU way not to get replaced, but it still avoids the MRU way. Since it's taking a guess, it is more dependent on software to see how good or bad this method will be for each situation. The benefit you get by taking worse efficiency is that the bit overhead is much less, needing n-1 bits for an n way set associative cache. In the case of our 4 ways, we only needed 3 bits opposed to the 8 bits in LRU, and an 8-way cache needs 7 instead of 24bits. This comes at a simpler design on hardware and more space for other data that isn't taken up by the PLRU itself.

Files Provided

Within the zip folder you will find the following files:

1. Homework04_AndrewSchmidt.pdf
2. Both_Output.txt | LRU_Output.txt | PLRU_Output.txt
3. Cache_Replacement_HW4_AndrewSchmidt.cpp