ECE552, Lab Assignment 4, Report: Data Caches

Date: November 24, 2020

Sebastian Wardzinski - 1004070676

Problem Statement

View comments in microbenchmark files for further context and details.

Question 1

For this test I had a cache configuration of dl1 (name): 2 (sets): 8 (block): 1 (associativity): l (replacement): 1 (prefetch flag). The main thing is that we don't want the data to fit in the cache as we iterate through the loops in the microbenchmark. Here I had two data arrays, both of which are much larger than the cache, one was made up of 'long long' elements (8B) and the other was made up of int elements (4B). In this microbenchmark file 3 test cases are covered:

1) Loop backwards through a data array of "long long" data (8 bytes each). Since we're going backwards the nextline prefetch misses every time. The elements are the same size as the blocks, so the cache itself also always misses. We get a miss rate of 0.9989.
2) Loop backwards through a data array of "int" data (4 bytes each). The nextline prefetcher still misses every time. However, since two elements fit in a single block, the cache itself hits half the time. We end up with a miss rate of 0.4995.
3) Loop forwards through a data array of "long long" data (although a similar result would be reached with the int array). The nextline prefetcher hits nearly every time, it only misses on the first element of the array (1/32 = 0.03125). This is almost exactly the miss rate we get, the exact value was 0.03134.

Question 2

When testing the stride prefetcher I had the same cache configuration as in question 1. Here I had one array, made up of 'long long' elements. Once again there were 3 cases covered:

1) Loop backwards through the data array (stride of -8 bytes). Since the stride is consistent, we get a very low miss rate (0.0323, which is ~1/32, one for each iteration of the outer loop). This is one of the major advantages of using a stride prefetcher, we can handle constant strides beyond those that are positive and less than a block of length (which is the extent that the next-line prefetcher can do).

2) Here we index into the array with varying strides (i*i has derivative 2i, which is not constant). This makes us miss every prefetch (0.9997); however, it is important to note that this is a difficult scenario for any prefetcher.
3) Here we see what happens when you break away from the stride every X iterations. Predictably, when we change the stride every other iteration, we end up in the no_pred state and don't make any prefetches (0.9997 miss rate). When we change the stride every 3rd element, we tend towards the state 'steady', and we just miss the prefetch for every 3rd element (0.3356). This pattern continues as we increase N, as we miss every 1/N element (where N > 2).

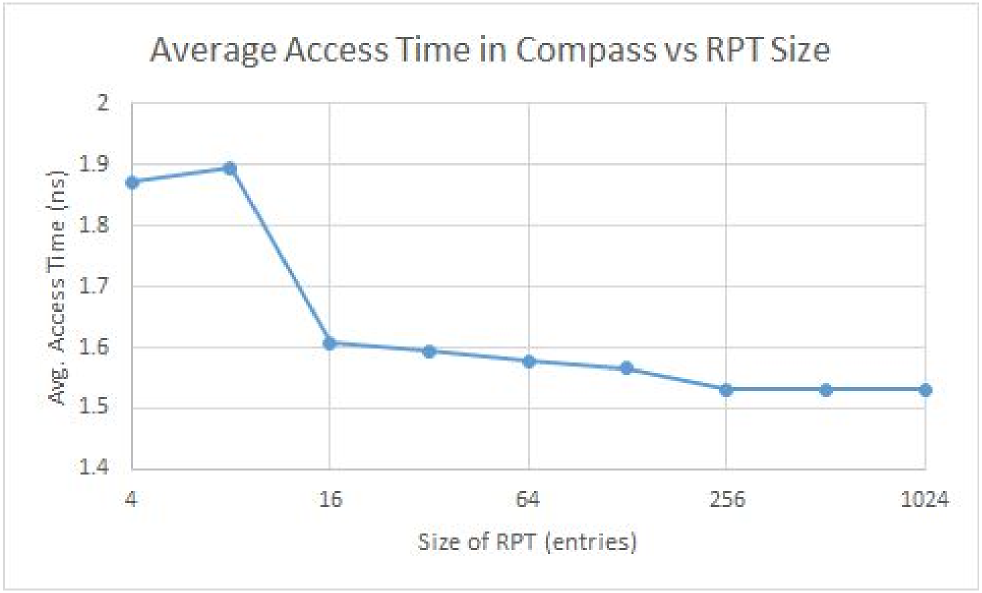
Question 3

Average access time = TaccessL1Data + %missL1Data * (TaccessL2 + %missL2 * TaccessMem)

Where: TaccessL1Data = 1, TaccessL2 = 10, TaccessMem = 100 (assuming units are 'cycles')

Config	(D)L1 Miss Rate	L2 Miss Rate	Average access time (cycles)
baseline	0.0416	0.1140	1 + 0.0416 * (10 + 0.114 * 100) = **1.890**
nextline	0.0419	0.0838	1 + 0.0419 * (10 + 0.0838 * 100) = **1.770**
stride	0.0385	0.0578	1 + 0.0385 * (10 + 0.0578 * 100) = **1.608**

Question 4



I chose average access time for the y-axis because I felt like it was the most encompassing and representative metric for the prefetch performance. I choose to use a log2 scale for the x-axis to better show the effect of size on avg. access time across multiple magnitudes.

Most of the gains are found early on, and we get within 5% of the best latency (that with sizes 512 & 1024) with just 16 entries in RPT. However, it is important to note this Access Time vs. Size curve is highly specific for whatever benchmark we use. The shape of this curve as well as the minimum avg. access time will be different depending on the program.

Question 5

It would've been convenient for me to have the statistics related to the average access time, that way I wouldn't need to have used formulas in excel to calculate them. However, for the software to be able to calculate this, the caches in the config files would require yet another parameter: access time.

Question 6

Since the open-ended prefetcher is very similar to the stride prefetcher (I suggest you read 'Open-Ended Prefetcher Implementation' before continuing here), it will behave the same way for all the test cases used in question 2. Here, the challenge I took on was to create a microbenchmark which had a different result for the stride and the open-ended predictors. Please look at the mbq6.c for full details.

This was deceptively hard, one of the only practical changes between the two is how fast they end up in the no_pred state. The open-ended prefetcher goes there when the stride doesn't match two times in a row, while the stride prefetcher needs 3 stride mismatches in a row to get there. So in my microbenchmark I made it reach a steady state (of stride 1), then I would have a stride of 2 for two iterations. Finally on the next cycle I would have a stride of zero. Since the stride prefetcher would be in the transient state at that point, it would prefetch (using a stride of 2) and replace the block which it is currently standing on. The open-ended prefetcher wouldn't replace the block and would instead have a hit.

Open-Ended Prefetcher Implementation

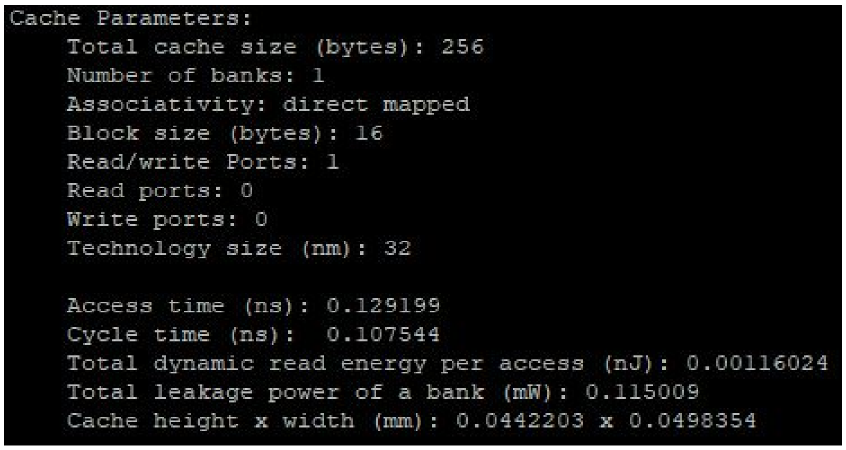
My open-ended prefetcher is a modification based on the stride prefetcher. The stride prefetcher itself gets an average miss rate of 2.25% across all three benchmarks. To allow you to fully appreciate the final prefetcher I will go through the steps of how I got to it:

1) First I increased size to 1024, this gave a miss rate of 2.12%.
2) I then played around with how selective the prefetcher is for actually making prefetches. When I made the prefetcher only prefetch in the STEADY state, it achieved 1.973%. When I made it prefetch for both STEADY & INIT , it achieved 1.963%.
3) I then reduced the size to 256 (to increase feasibility), and the accuracy dropped to 1.970%. I tried to change the transitions of the TRANSIENT state so that it changes to the INIT state if the stride matches (as opposed to going to 'steady'), so that it acts like a regular 2-bit saturating counter. This didn't help and increased the miss rate to 1.973%.
4) Finally I thought, maybe the init state itself isn't necessary, I removed the init state and adjusted the transitions and initialization accordingly. This worked and dropped the miss rate down to 1.963% the lowest it reached with an RPT of size 256.
5) As a flex, I then lowered the size of the RPT back to 16 entries, which was the original amount used by the stride predictor. I managed to get 2.02%, meaning that I reduced the miss rate by 11.4% (from 2.25% to 2.02%) only by optimizing the state machine and without making any changes to the RPT table or the hashing required to index into it.

One of the pros of having such a small RPT table is that its access times will be low, which is especially important if accessing the RPT table is part of the critical path. The RPT table is best represented by a RAM structure.

Both the tag and prev_addr have to be integers in order to keep the aliasing at a minimum, because of this each RPT entry would require at least 10 bytes. This means that 16 bytes would have to be allocated for each entry. Nevertheless, our table would take up 256 bytes, with block size 16, associativity 1, and a bus of 128 bits (assuming you want access to all of the fields in one read/write). All-in-all, it will be a lightweight memory module with a cycle and access time that is many times faster than that of modern processors. See Figure 1 for the CACTI analysis.

Figure 1: Time, Energy, and Area parameters of RPT RAM module for the stride & open-ended predictor


```
Cache Parameters:
    Total cache size (bytes): 256
    Number of banks: 1
    Associativity: direct mapped
    Block size (bytes): 16
    Read/write Ports: 1
    Read ports: 0
    Write ports: 0
    Technology size (nm): 32

    Access time (ns): 0.129199
    Cycle time (ns):  0.107544
    Total dynamic read energy per access (nJ): 0.00116024
    Total leakage power of a bank (mW): 0.115009
    Cache height x width (mm): 0.0442203 x 0.0498354
```