ECE552 – Lab Assignment 4 Report: Data Caches

Andreea Varlan & Ben Pak

Question 1

q1.cfg: nsets: 64; block_size = 8 bytes; associativity = 1 way

Using an **access step size of 2**, we access an array of integers (4 bytes each element) at every 8 bytes which is the configured block size. We always access the very next block, which is exactly what the next line prefetcher predicts, and thus our L1 data cache miss rate is **0.0%**.

As expected, when accessing every 12 bytes, (with a step size of 3 in the array), the next line prefetcher misses half of the time.

arr[0]	arr[1]
arr[2]	arr[3]
arr[4]	arr[5]
arr[6]	arr[7]
arr[8]	arr[9]

Highlighted on the left are the elements within the mbq1 array that are accessed on each iteration (every third one). When we access addr of arr[0], the nextline prefetcher will fetch the next block which includes arr[3] addr, thus it is a HIT. But after accessing arr[3], we will next access arr[6] which is not found in the next prefetched block, thus it will be a MISS. This cycle repeats for the entirety of the array, causing a MISS every other cycle, thus the miss rate is **49.99%.**

Question 2

Q2.cfg: nsets: 64; block_size = 8 bytes; associativity = 1 way; RPT_size =16

Using a similar setup as for the next-line prefetcher, when setting a **constant** stride (of 10, for example) every 10^{th} element of the array would be accessed. This constant stride gives us a low miss rate in the L1 data cache of just **7.8%,** which accounts for the iterations of the large outer loop.

In the next test, we change the stride after every single access to the array in a non-constant way by increasing the step size by 10 every time. Through this we achieve exactly what we expected, a large miss rate value of **99.98%** as the stride prefetcher is not able to pick up on any address access patterns.

For the final test, I thought to only change the stride pattern every n elements (in my example I chose every 22 elements). Since I am iterating over a large array (100000 elements), I still expected to see a high miss rate, but visibly smaller than in the completely non-constant stride example above. Indeed, the miss rate dropped but remained high at **94.76%**, reconfirming the functionality of the stride prefetcher.

Question 3

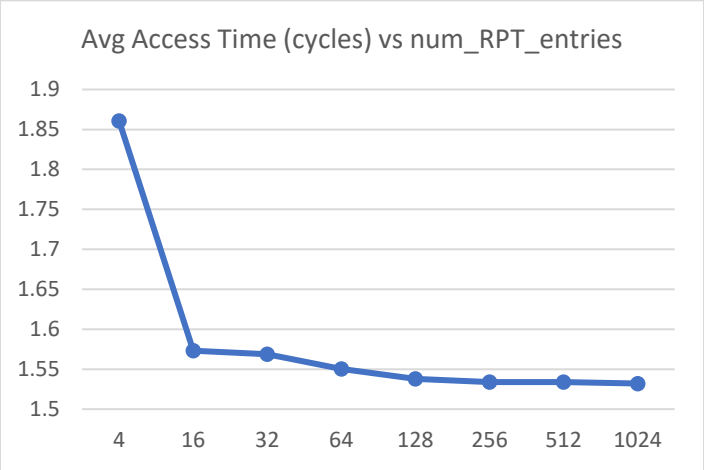
Using benchmark *compress*, and given the following hit times:

T[access−L1Data] = 1; T[access−L2] = 10; T[hit-Memory] = 100.

Avg Access Time = T[access−L1Data] + %miss[L1Data] * (T[access-L2] + %miss[L2Data] * T[access-Memory])

Config	L1 Miss Rate (%)	L2 Miss Rate (%)	Average Access Time (cycles)
Baseline	4.16	11.4	1 + 0.0416 + (10 + 0.114 * 100) = **1.890**
Next-line	4.19	8.38	1 + 0.0419 * (10 + 0.0838 * 100) = **1.770**
Stride	3.78	5.16	1 + 0.0378 * (10 + 0.0516 * 100) = **1.573**

Question 4



RPT_entries	Access_time	dl1_miss_rate	ul2_miss_rate
4	1.860458	0.0422	0.1039
16	1.573048	0.0378	0.0516
32	1.568888	0.0376	0.0513
64	1.550188	0.0372	0.0479
128	1.53798	0.037	0.0454
256	1.533869	0.0371	0.0439
512	1.533869	0.0371	0.0439
1024	1.53206	0.037	0.0438

Using the same Avg Access Time equation as in question 3, I was able to observe that there is a significant improvement of the stride prefetcher when increasing the number of RPT entries past 16, and that it reaches a performance plateau in terms of access time after 128-256 entries.

Question 5

I think the average access time does prove to be the most encompassing performance feature to observe over changes in different configuration parameters. I know that we are given all the information to be able to derive it ourselves, but it would improve efficiency of performance review to have it as a precalculated statistic in sim-cache, from where we could also easily update the access hit times that are used in the latency calculation.

Question 6

Since my open-ended prefetcher is a modified stride prefetcher, creating test cases that differ from mbq2 has proven quite difficult.

Using the same test cases, I observed that for the constant stride, as well as for the completely non-constant stride, there was no change in the miss rate compared to the regular stride prefetcher.

For a changing constant stride at every 22 elements accessed in the array, there was a change in the miss rate value, specifically it increased from 94.76% to **97.78%.** This is contrary to what I initially expected, since across the given benchmarks the miss rate was obviously lower than for the normal stride prefetcher. To reason about it, I believe that since we are only prefetching in the STEADY state, useful prefetches that might have occurred in the other stages (specifically in the INIT state) do not occur, and thus increase the miss rate.

Open-ended prefetcher

I realized that the performance of the stride prefetcher I implemented is a miss rate of **2.2%** which is quite close to the desired miss rate for the open-ended prefetcher. Thus, I chose to first explore ways of improving the stride prefetcher by changing different parameters or its structure.

From the insights from question 4, I increased the number of RPT entries to 1024. However, I did not observe any change in the performance of the strider.

I then considered at which stages the prefetcher performs a prefetch. Instead of prefetching at any state other than NO-PRED, I only prefetched at the STEADY state. I was surprised to see that the miss rate dropped way down to **1.9833**%. Then I tried to prefetch only at the INIT state, as well as only at the TRANSIENT stage. This yielded a miss rate of 2.27% and 2.37% for the INIT and TRANSIENT respectively. Trying a combination of prefetches occurring in both the STEADY and the INIT state also did not yield a more satisfactory result than for solely in the STEADY state.

With the prefetcher set to only prefetch on the STEADY state, I tested the different block replacement policies. Using the FIFO method yielded an avg miss rate of **2.24%**, while using Random replacement resulted in a **2.33%** miss rate. Thus, the optimum replacement policy for our prefetcher is the LRU method.

After reading a paper on Markov Chains, I was curious to see what effects an added data structure, such as a missed history table, would have on the miss rate. I tried implementing a wrap-around buffer that stores the addresses on which a cache miss occurs (adding of the elements in the buffer happens in the cache_access function when the cache block is not found). Whenever there would not be a tag match in the RPT, on top of adding that instruction to the RPT, I would prefetch from the missed history buffer the latest address. However, I was not able to test the functionality and performance of this feature due to an issue with the setting of a variable related to this structure in cache_create (specifically an INT that stores the index

representing the head of the structure). Every time I would initialize it in cache_create, as soon as the for loop for linking the data blocks would begin, the variable would have a bogus value, leading me to believe that it was overwritten in memory by the new allocation of blocks in a set.

As for the CACTI file, since I am using only the RPT as the main structure, it having 1024 entries, and each entry being made of 2 md_addr_t addreses, 1 int and an enum that designates the rpt state, the total RAM size needed is:

1024 * (2*4 bytes + 4 bytes + 8 bytes) = 20480 bytes

Using a pureRAM cacti configuration file, we get the following statistics:

```
Cache Parameters:
    Total cache size (bytes): 20480
    Number of banks: 1
    Associativity: direct mapped
    Block size (bytes): 20
    Read/write Ports: 1
    Read ports: 0
    Write ports: 0
    Technology size (nm): 32

    Access time (ns): 0.363629
    Cycle time (ns):  0.469473
    Total dynamic read energy per access (nJ): 0.00496718
    Total leakage power of a bank (mW): 6.50934
    Cache height x width (mm): 0.164661 x 0.180845
```

Since the RPT size seemed to not effect greatly the miss rate, especially when tested on the given benchmarks, it would be more feasible and efficient to use an RPT of size 128 (first value that reaches a plateau of low avg access time in q4), which would reduce the time, energy and area parameters, as shown in the CACTI statistics below:

128 * (2*4 bytes + 4 bytes + 8 bytes) = 2560 bytes

```
Cache Parameters:
    Total cache size (bytes): 2560
    Number of banks: 1
    Associativity: direct mapped
    Block size (bytes): 20
    Read/write Ports: 1
    Read ports: 0
    Write ports: 0
    Technology size (nm): 32

    Access time (ns): 0.220285
    Cycle time (ns):  0.162409
    Total dynamic read energy per access (nJ): 0.00100105
    Total leakage power of a bank (mW): 1.02791
    Cache height x width (mm): 0.0850017 x 0.0512217
```

Statement of work: Ben has written the first draft of the microbenchmarks. Andreea has implemented everything else.